



# SBM

# SEBM

Sistemes Basats en Microprocessadors

Sistemes Electrònics Basats en  
Microprocessadors

Manual de Pràctiques

Manel Domínguez, Vicente Jiménez,  
Daniel Bardés, Clemente Pol.

## Historial del document

Data	Versió	Responsable	Descripció
	1.0	V. Jiménez	Versió inicial
	1.1	V. Jiménez	Puntuals pàg. 5, 37, 38, 57, 59 Descripció IDR (p100)
27/6/2013	1.2	V. Jiménez	Modificació completa de P1 per adaptar-la a STM32v2
	1.3	V. Jiménez	P1: Canvi "vista" per "perspectiva" Vincle a creació d'un nou Workspace C2: Es demana un decimal de precisió a Tmeas Indicació addicional sobre TSVREFE
	1.4	V. Jiménez	P1: Correcció sobre visualització de variables P3: Petites clarificacions P4: Figura diagrama de flux p92
	1.5	D. Bardés	Múltiples petites correccions, ortogràfiques i d'altres.
26/6/2014	1.6	V. Jiménez	Adaptacions per canvis en eines de laboratori S'inclou un segon temporitzador per retards de $\mu$ s
30/6/2015	1.7	V. Jiménez	Errors pag 14, 19, 35 Explicacions sobre l'arrencada associada a <i>board.h</i> Pag 99: Indicació que C1 es obligatòria en SEBM Annex A1 - Cable de comunicacions EXT1
13/7/2015	1.8	V. Jiménez	Conversió a anglès del fitxers font Us del fitxer <i>labBoard12.h</i> Referències a <i>stm32f4xx.h</i>

# Contingut

Introducció .....	4
I1 - Objectius i organització de les pràctiques .....	4
I2 - Criteris d'avaluació .....	6
I3 - Descripció de la placa de laboratori .....	8
I4 - Descripció de les eines de software.....	12
I5 - Manuals addicionals .....	13
Pràctiques bàsiques .....	14
P1 - Introducció a Eclipse .....	14
P2 - Accés al'LCD .....	45
P3 – Acceleròmetre 3D .....	68
P4 – Interrupcions i mesures de temporització .....	86
Pràctiques complementàries.....	103
C1 - Lectura del teclat .....	104
C2 - Convertidor A/D .....	113
C3 - Encoder .....	124
Pràctiques amb RTOS .....	133
R1 - Introducció a ChibiOS/RT .....	134
R2 - Compartició d'elements comuns .....	147
Annexes.....	156
A1 - Cable de Comunicacions EXT1.....	156

# Introducció

## I1 - Objectius i organització de les pràctiques

Aquestes pràctiques prenenen donar suport pràctic i complementar els continguts de teoria de les assignatures de Sistemes Basats en Microprocessadors (SBM) i Sistemes Electrònics Basats en Microprocessadors (SEBM). En primer lloc el suport és pràctic per que es pretén fer aplicació d'alguns dels continguts de teoria per reforçar l'aprenentatge. En segon lloc, el suport es complementari per que alguns dels continguts de pràctiques no tenen reflex a les classes de teoria donada la dificultat que té explicar alguns elements dels sistemes basats en microprocessador des d'una perspectiva purament teòrica.

Els objectius que es prenenen assolir a les pràctiques són:

- Introduir un entorn de programació/depuració basat en Eclipse
- Entendre i programar ports GPIO d'entrada/sortida
- Entendre i programar alguns altres perifèrics interns del microcontrolador
- Desenvolupar comunicacions a baix nivell amb perifèrics externs
- Desenvolupar el codi associat a la gestió d'interrupcions
- Analitzar la temporització de comunicacions amb perifèrics
- Introduir la programació amb un sistema operatiu en temps real (RTOS)

El desenvolupament de sistemes basats en microcontroladors té una corba d'aprenentatge que, per a algunes persones, pot ser molt forta. Tanmateix es vol que els estudiants aprofitin tant com puguin el seu temps de pràctiques. Es per això que les pràctiques estan dividides en una part obligatòria i un conjunt de elements opcionals. Per completar les pràctiques es imprescindible completar la part obligatòria. Acabada aquesta part, l'estudiant pot realitzar elements opcionals per millorar la seva qualificació.

Les pràctiques estan molt guiades i orientades a objectius molt concrets. Aquest manual conté pràcticament tota informació necessària per completar-les. No obstant, a fi de complementar la informació sobre el funcionament de la placa sobre la que es realitzen i de donar suport a treballs optional oberts, es subministren també un conjunt de manuals del microcontrolador de la placa i els principals perifèrics que fem servir.

A l'assignatura de **SBM** les pràctiques estan organitzades en 6 sessions de 2h cadascuna. Les sessions de pràctiques es distribueixen en setmanes alternes A/B. Per tant, cada estudiant, disposarà d'una sessió de 2h de laboratori cada dues setmanes. Dins de les 6 sessions es obligatori completar els 4 apartats obligatoris P1, P2, P3 i P4.

En cas de completar tots els apartats obligatoris, l'estudiant pot optar per desenvolupar altres apartats opcionals per obtenir una millor qualificació a l'assignatura.

A la assignatura de **SEBM** les pràctiques estan organitzades en 13 sessions de 2h cadascuna (una cada setmana lectiva). Dins de les 13 sessions és obligatori completar els apartats obligatoris P1, P2, P3, P4, C1, R1 i R2.

En cas de completar tots els apartats obligatoris, l'estudiant pot optar per desenvolupar altres apartats opcionals com C2 i C3 per obtenir una millor qualificació a l'assignatura.

Al principi de cada pràctica s'indica quins són els estudis previs que s'han de lliurar en començar-la i quins són els resultats que s'han d'obtenir en acabar-la.

Els punts on heu d'interactuar amb l'ordinador acostumen a estar marcats amb el símbol llevat de zones d'interacció molt extenses on ja s'entén del context.

Les interaccions amb la placa de pràctiques acostumen a estar marcades amb el símbol .

En alguns punts es suggeriran funcionalitats opcionals que no són estrictament necessàries però milloren el codi generat. Aquestes s'assenyalaran amb la icona .

Els llocs on heu de vigilar per que és fàcil cometre algun error o generar *bugs* al codi de les pràctiques estan senyalats amb la icona .

Finalment, podreu trobar la icona on es dona informació lateral addicional no imprescindible pel desenvolupament de les pràctiques.

## I2 - Criteris d'avaluació

L'avaluació de les pràctiques es fa tenint en compte els següents criteris:

### a) Assoliment dels objectius específics demanats (30%)

Dins d'aquest apartat s'avalua que s'hagin assolit els objectius concrets que es demanen als apartats obligatoris. En concret s'avalua que es compleixi la funcionalitat sol·licitada i que es faci a un ritme adequat.

### b) Qualitat de la programació (25%)

Dins d'aquest apartat s'avalua la qualitat dels programes realitzats. En concret es valora:

- Claredat del programa (Ordre, comentaris, etc...)
- Potencialitat del programes desenvolupats. És a dir, la capacitat del codi d'encabir funcionalitats mes grans que les mínimes demandades.
- Flexibilitat i escalabilitat dels programes. És a dir, la capacitat de poder ampliar les funcionalitats del programa sense haver de modificar massa el codi ja desenvolupat.
- No fer servir un número excessiu de recursos (Variables, codi, etc...)

### c) Innovació i originalitat (20%)

Aquest apartat està lligat principalment a la realització dels apartats opcionals. Tanmateix, un desenvolupament d'alta qualitat i/o originalitat dins dels apartats obligatoris també es pot comptar, de manera excepcional, dins d'aquest apartat.

### d) Memòria final (10%)

La memòria final ha de contenir tots els resultats que es demanen als apartats obligatoris i opcionals que es facin. Aquests resultats, no obstant, puntuen dins dels apartats anteriors. El principal pes del apartat d) no és tant els resultats com la capacitat de l'estudiant de exposar amb claredat els objectius i resultats obtinguts. També es valora la capacitat de justificar les decisions que s'han pres en el desenvolupament dels programes.

La memòria ha de ser auto-continguda. És a dir, ha de poder ser llegida per qualsevol persona que tingui bons coneixements sobre la assignatura i els elements de la placa de laboratori però que no tingui accés als manuals de pràctiques.

La memòria final ha de contenir obligatòriament el codi de tots els programes desenvolupats.

### e) Control de pràctiques (15%)

El control de pràctiques és, en principi, l'únic element avaluador individual del laboratori. En aquest control es pretén valorar la interiorització que ha fet l'estudiant sobre el funcionament dels elements que s'han fet servir en els apartats obligatoris.

#### **Originalitat del material**

Tot el material entregat per l'estudiant, incloent estudis previs i codi ha de ser original. L'únic codi no original que es pot entregar es el facilitat amb el material de pràctiques.

En cas de detecció de entrega de material fraudulent, s'aplicarà la normativa de l'escola. En el moment de la redacció d'aquest document aquesta normativa implica el suspens de l'assignatura amb nota de zero i la possible apertura d'un procès disciplinari.

S'entén explícitament com no original qualsevol contingut obtingut d'un estudiant aliè al grup que presenta el treball.

## I3 - Descripció de la placa de laboratori

La figura següent mostra la placa de pràctiques. La placa s'ha desenvolupat al voltant d'una altra placa STM32F4 Discovery (placa de la dreta amb connector USB) a la que s'hi han afegit alguns perifèrics addicionals com un LCD i un teclat entre altres.



A continuació descriurem la placa Discovery per després descriure, breument, els perifèrics que s'hi han afegit.

### Placa STM32F4 Discovery

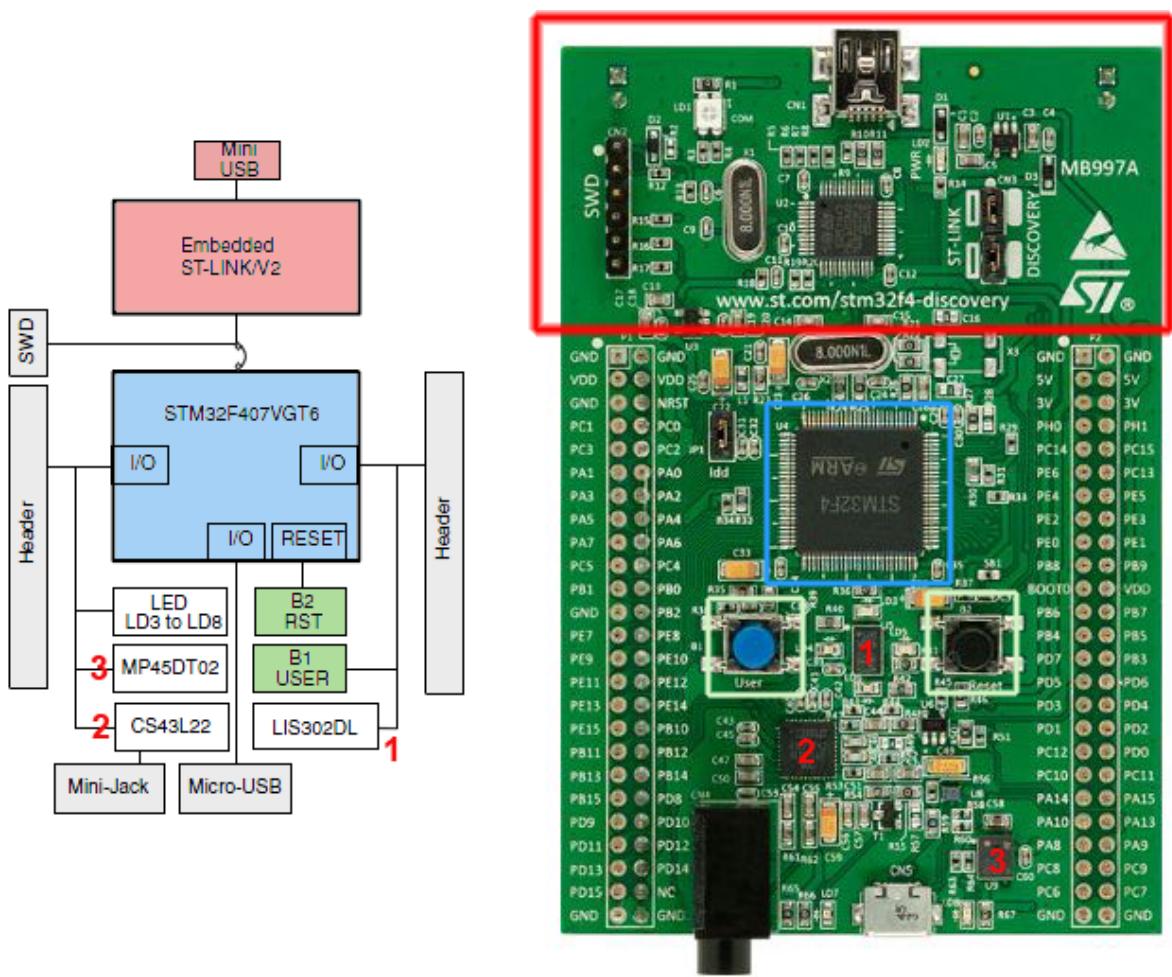
La placa STM32F4 *Discovery* pertany a la família de plaques d'avaluació *Discovery* de la companyia ST Microelectronics (<http://www.st.com>). Aquesta companyia facilita les plaques *Discovery* a molt baix cost per incentivar que els seus clients potencials puguin familiaritzar-se amb els seus microcontroladors. Per fer-se una idea de que vol dir baix cost, totes les plaques de la família *Discovery* es troben a una forquilla típica de preus entre 10€ i 15€.

El nucli de la placa es un xip microcontrolador STM32F407VGT6 amb encapsulat LQFP (Low-profile Flat Quad Package) de 100 pins (a la figura següent, al centre envoltat per un quadre blau) que inclou 1MByte de memòria flash i 192 KByte de memòria SRAM i és capaç d'operar a una freqüència interna de 168MHz. El microcontrolador inclou un nucli processador ARM Cortex M4de 32 bits envoltat per un conjunt de perifèrics entre els que hi trobem:

- 12 Temporitzadors de 16 bits
- 3 Convertidors Analògic → Digital (ADC) de 12 bits
- 2 Convertidors Digital → Analògic (DAC) de 12 bits
- 4 Blocs de comunicació Asíncrona (USART)
- 3 Blocs de comunicació Síncrona I2C
- 3 Blocs de comunicació Síncrona SPI/I2S
- 2 Interfícies Controller Area Network (CAN)

- 1 Interfície USB OTG
- 1 Interfície Ethernet
- 1 Generador de nombres aleatoris
- 2 Watchdogs

A la part superior de la placa, fins on hi ha el text de la direcció d'internet, s'hi troba el programador/emulador ST-LINK V2. Aquesta part de la placa es l'encarregada de connectar amb el PC mitjançant un connector Mini-USB. L'emulador fa d'interfície amb els pins de programació/depuració JTAG del xip. La implementació física de l'emulador es fa mitjançant un microcontrolador STM32F103C8T6, just a sota del connector USB, independent del microcontrolador principal de la placa.



Placa STM32F4 Discovery

A part dels perifèrics interns del processador, inclosos a la placa STM32F4 *Discovery* trobem també altres perifèrics:

- Un DAC extern d'àudio CS43L22, (número 2 a la figura), amb entrades analògiques, entrada digital I2S i entrada de control I2C. La sortida es troba connectada a un jack d'àudio estèreo estàndard de 3.5mm .

- Un micròfon MEMS MP45DT02, (número3 a la figura), amb sortida digital PDM.
- Un acceleròmetre MEMS LIS302DL, (número 1 a la figura) de tres eixos amb interfície SPI.
- 4 Leds (Taronja, Verd, Vermell i Blau) que envolten el xip acceleròmetre.
- 1 Polsador d'usuari (B1)
- 1 Polsador de Reset (B2)

A la part inferior de la placa, a part del jack d'àudio, tenim també un connector USB que connecta amb la interfície USB-OTG del microcontrolador que permet, per exemple, accedir a *pendrives* des del microcontrolador.

Als dos costats de la placa trobem els connectors de 50 pins cadascun que permeten accedir a gairebé tots els pins del microcontrolador. Tots els pins inclouen els noms de les senyals que connecten. D'aquesta manera es simplifica la connexió dels senyals al analitzador lògic.

### Perifèrics addicionals

La placa STM32F4 Discovery va muntada a sobre d'una placa PCB que inclou alguns perifèrics addicionals.

- Un mòdul LCD alfanumèric de 2 files de 16 caràcters amb il·luminació LED. L'ajust de contrast es fa mitjançant el potenciòmetre de color verd que es troba a sota.
- Un teclat matricial de 16 tecles.
- Un potenciòmetre d'usuari que pot ser llegit per un dels convertidors A/D del microcontrolador. Es troba sota el potenciòmetre de contrast i té color groc
- Un codificador de quadratura que es troba sota el potenciòmetre d'usuari i que té color blau.
- Un polsador de RESET connectat en paral·lel amb el de la placa *Discovery* que es troba a dalt a la dreta de la placa.
- Un polsador d'USUARI connectat en paral·lel amb el de la placa *Discovery* que es troba sota el polsador de RESET.
- Dos connectors per perifèrics externs EXT1 i EXT2

Per tal de no fer malbé els polsadors de RESET i d'USUARI que hi ha a la placa *Discovery*, es recomana fer servir els polsadors alternatius que es troben a la part dreta de la placa de pràctiques.

La descripció de les connexions entre el microcontrolador i els perifèrics, tant els de la placa *Discovery* com els addicionals es troba al document "[Esquemes.pdf](#)".

### Descripció dels connectors

La placa disposa de 5 blocs de connectors. Tres connecten la placa general amb la placa de microcontrolador i els perifèrics inclosos i 2 permeten connexions a perifèrics externs.

## CN1 i CN2 - Placa STM32F4 *Discovery*

Aquests 2 connectors de 2 fileres de 25 pins cadascun comuniquen la placa STM32F4 *Discovery* amb la resta del sistema. La descripció dels pins del connector la podeu trobar al document "[Discovery.pdf](#)".

## CN3 - LCD

Aquest connector comunica la placa de pràctiques amb el mòdul LCD. La tensió d'alimentació del LCD es pot escollir entre +5V i +3V mitjançant un pont. A la versió actual de la placa, Vdd és de +5V. Observeu que els pins 7 a 10 no estan connectats. La distribució de pins, de esquerra a dreta, és la que es troba a la taula següent.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GND	Vdd	Vo	RS	GND	E				DB4	DB5	DB6	DB7	+5V	Led-	

## CN4 - Teclat

Aquest connector comunica la placa de pràctiques el teclat matricial. El teclat matricial té 4 fileres R1..R4 i 4 columnes C1..C4. La distribució de pins de esquerra a dreta és la que es mostra a la següent taula:

1	2	3	4	5	6	7	8
R1	R2	R3	R4	C1	C2	C3	C4

## CN5 - Extensió 1

Aquest connector permet ampliar la funcionalitat de la placa amb perifèrics externs. La següent taula mostra de dalt a baix la distribució de pins, les funcions principals que tenen(GPIO o alimentació) i les principals funcionalitats alternatives que poden tenir (Connexions directes a perifèrics interns). Les principals funcions especials són a les línies PA8/PC9 que incorporen un bus I2C i a la línia PB7 que té una USART de recepció.

Pin	Funció	Funcions alternatives
1	GND	
2	+5V	
3	+3V	
4	No connectat, es fa servir per alineació	
5	PC15	
6	PC13	
7	PE5	TIM9-CH1
8	PB7	<b>TIM4-CH2 / USART1-RX</b>
9	PB5	TIM3-CH2
10	PA8	MCO1 / TIM1-CH1 / <b>I2C3-SCL</b>
11	PC9	MCO2 / TIM3-CH4 / TIM8-CH4 / <b>I2C3-SDA</b>

## CN6 - Extensió 2

Aquest connector, igual que l'anterior, permet ampliar la funcionalitat de la placa amb perifèrics externs. La següent taula mostra d'esquerra a dreta la distribució de pins, les funcions principals que tenen (GPIO o alimentació) i les principals funcionalitats alternatives que poden tenir (connexions directes a perifèrics interns). Les principals funcions especials son a les línies PB15,/PB13/PB12/PB14 que incorporen un SPI o un bus CAN i a les línies PC6/PC11 que incorporen una USART de transmissió i una altra de recepció.

Pin	Funció	Funcions alternatives
1	PD10	
2	PB15	RTC-50Hz / TIM1-CH3N / TIM8-CH3N / TIM12-CH2 / <b>SPI2-MOSI</b>
3	PB13	TIM1-CH1N / <b>CAN2-TX</b> / <b>SPI2-SCK</b>
4	PB14	TIM1-CH2N / TIM8-CH2N / TIM12-CH1 / <b>SPI2 - MISO</b>
5	PB12	TIM1-BKIN / <b>CAN2-RX</b> / <b>SPI2-NSS</b>
6	+3V	
7	PC8	TIM3-CH3 / TIM8 - CH3 / USART6-CK
8	PC6	TIM3-CH1 / TIM8 - CH1 / USART6-TX
9	GND	
10	No connectat, es fa servir per alineació	
11	PA15	TIM2-CH1 / TIM2-ETR
12	PC11	<b>USART3-RX / UART4-RX</b>

Els connectors d'extensió 1 i 2 tenen cadascun un pin sense connectar. Aquest pin no té forat de manera que el connector d'extensió només es pot introduir en l'orientació correcta.

## I4 - Descripció de les eines de software

Una vegada descrita la placa amb que treballarem, descriurem breument l'entorn de treball. Les pràctiques es desenvolupen fonamentalment amb eines de codi obert. L'eina principal de treball serà l'entorn Eclipse (versió *Juno*). Aquest es un entorn de programació desenvolupat amb Java que uneix les eines d'edició, compilació i depuració.

Per compilar els programes farem les eines GNU ARM. Aquest és un conjunt d'eines GNU de codi obert que permeten compilar i enllaçar programes per processadors ARM. Es tracta d'un compilador creuat. És a dir, que compilem en un PC per crear codi ARM que el PC no pot executar i que s'ha de enviar a la placa per que l'executi.

Al nostre codi hi afegirem el sistema operatiu ChibiOS/RT. Per programar el microcontrolador, de fet, no fa falta cap sistema operatiu, i el treball amb sistemes operatius excedeix els objectius de les pràctiques de SEBM/SBM, però el codi del sistema ens permetrà simplificar la inicialització del microcontrolador. Addicionalment, si volem desenvolupar un programa complex, el sistema operatiu ens permetrà simplificar la interacció amb el hardware.

Als elements anteriors hi afegirem els *drivers* USB de comunicació amb la placa i el servidor GDB que establirà la connexió necessària entre Eclipse i la placa per fer la càrrega i depuració del programa.

## I5 - Manuals addicionals

Més enllà d'aquest manual podreu trobar els següents manuals complementaris:

<a href="#">Discovery.pdf</a>	Manual de la placa STM32F4 Discovery
<a href="#">STM32F4 Reference.pdf</a>	Manual de la família STM32F4xx
<a href="#">STM32F407 Datasheet.pdf</a>	Manual específic de STM32F407
<a href="#">Esquemes.pdf</a>	Esquemes elèctrics de la placa
<a href="#">LIS302DL.pdf</a>	Manual del Acceleròmetre 3D
<a href="#">CS43L22.pdf</a>	Manual del DAC de àudio
<a href="#">MP45DT02.pdf</a>	Manual del micròfon
<a href="#">LCD MC1602C8.pdf</a>	Especificacions de la pantalla LCD
<a href="#">HD44780U Driver.pdf</a>	Manual del <i>driver</i> que fa servir el LCD

A part d'aquests manuals, les següents direccions d'internet poden ser també d'interès:

Direcció web de STM per la placa STM32F4 Discovery

<http://www.st.com/internet/evalboard/product/252419.jsp>

Direcció web de Eclipse

<http://www.eclipse.org>

Direcció web de GNU ARM

<https://launchpad.net/gcc-arm-embedded/>

Direcció web de ChibiOS/RT

<http://chibios.org>

### ① Informació sobre els vincles

Per què funcionin els vincles a documents PDF, els documents han de trobar-se a la mateixa carpeta que aquest manual de pràctiques.

Acrobat, per defecte, obre els vincles com els que es troben en aquesta pàgina dins de la finestra actual. Això fa que es tanqui el document origen, en aquest cas el manual de pràctiques.

Si voleu canviar aquest comportament, seleccioneu les **preferències** al **menú d'edició** i desseleccioneu, dins de la categoria de **Documents**, l'opció que permet obrir vincles entre documents a la mateixa finestra.

# Pràctiques bàsiques

Aquest primer bloc de pràctiques és obligatori tant per l'assignatura de SBM com per SEBM. Tots els estudiants han de desenvolupar la feina que es demana en aquest bloc per tenir una avaluació positiva de les pràctiques.

## P1 - Introducció a Eclipse

En aquesta primera pràctica es vol introduir l'estudiant a l'entorn de treball Eclipse. Això inclou copiar al directori del usuari els fitxers necessaris, arrencar i configurar l'entorn Eclipse, crear un nou projecte, compilar-lo i provar-lo. Al final de la pràctica es demanarà fer una petita modificació del programa subministrat com a exemple.

### Objectius de la pràctica:

- Introduir l'ús de l'eina Eclipse.
- Entendre el funcionament dels ports de sortida del microcontrolador.

### Estudis Previs:

- No hi ha, però s'ha de llegir l'enunciat de la pràctica abans d'anar al laboratori.

### Resultats:

- Crear i fer funcionar la funció **ledSequence()** que es descriu a P1.8

### P1.1 Verificació de la placa

El primer que hem de fer en començar una sessió de laboratori es verificar la placa que farem servir. La placa conté alguns elements delicats i s'ha de tractar amb cura. Tot i això sempre es poden produir problemes o fallades. És per això que abans de començar a treballar amb la placa convé verificar el seu funcionament.

Al laboratori farem córrer el nostre codi a la RAM del microcontrolador. Dins la memòria Flash no volàtil hauria d'haver sempre el programa de test que permet verificar el bon funcionament de la placa.

- ⚡ Connecteu la placa mitjançant un cable micro-USB - USB a un port lliure de l'ordinador. Verifiqueu que a la part de dalt de la placa *Discovery* s'encenen dos leds vermells (LD1 i LD2) de manera contínua.

A la pantalla hauria de sortir un text a la primera línia que indica que tenim carregat el programa de test "BOARD TEST" i la seva versió. Si no surt aquest missatge proveu de pitjar el botó de *reset* (a dalt a la dreta) o moure el potenciómetre verd de contrast del LCD.

Si tot i això no arrenca el programa de test, vol dir que no està carregat a la placa. En aquest cas executarem el programa **STM32 BoardTest** que té un vincle al escriptori del ordinador. Aquest programa permet reescriure la memòria Flash de la placa amb el codi de test. Després de carregar el codi pot caldre pitjar la tecla *reset* per arrencar el programa.

- ⚡ Un cop carregat el programa, s'executarà generant una seqüència cíclica de tests. Per passar de cada test al següent polsarem el pulsador d'USUARI (USR). Recordeu que si no es veu res al LCD, verifiqueu l'ajust de contrast amb el potenciómetre de dalt de color verd.

La seqüència de test té, al menys, els següents elements:

- Missatge d'inici
- Encesa de la llum del LCD
- Test dels LEDs de la placa (Els quatre han de fer pampallugues)
- Test del teclat (Ha de mostrar quina tecla possem)
- Test de l'acceleròmetre (Ha de ser sensible a l'inclinació de la placa)
- Test del potenciómetre (Ha de variar en girar-lo)
- Test del 'encoder' (Ha de variar en girar-lo)

A continuació venen els tests dels ports d'estensió 1 i 2. Per provar aquests ports s'han de connectar unes matrius de resistències específiques que es forneixen. No us preocupeu, per tant, si fallen aquests tests.

- Test del port d'estensió 1
- Test del port d'estensió 2

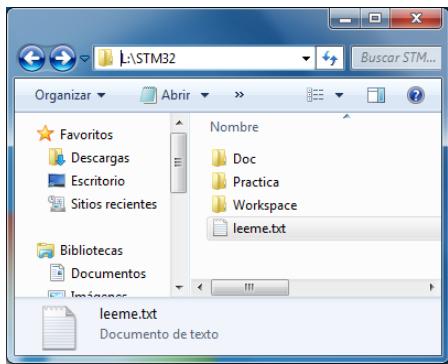
Pels últims tests s'ha de posar un auricular al connector jack de 3.5mm. Aquests tests no cal comprovar-los si no s'ha de fer cap apartat opcional d'àudio. El subsistema d'àudio té un temps d'arrencada, per tant, cal esperar uns segons per que funcioni el primer test. Si pareu l'espera amb el botó d'usuari, saltarem el test d'àudio.

- Test del micròfon (S'ha de sentir el so del micròfon per l'auricular)
- Test del DAC (S'ha de sentir una seqüència de tons per l'auricular)

## P1.2 Posada en marxa d'Eclipse

El procediment de posada en marxa d'un nou projecte a Eclipse és una mica complex. A les pràctiques ho hem simplificat donant-vos un projecte ja preconfigurat. Si voleu saber tots els passos que calen per crear un nou projecte podeu consultar el document "[Creació d'un Workspace.pdf](#)"

- Per començar a treballar amb Eclipse, obriu el fitxer **Practica-P1.zip** i arrossegueu el seu contingut al vostre directori d'usuari L:

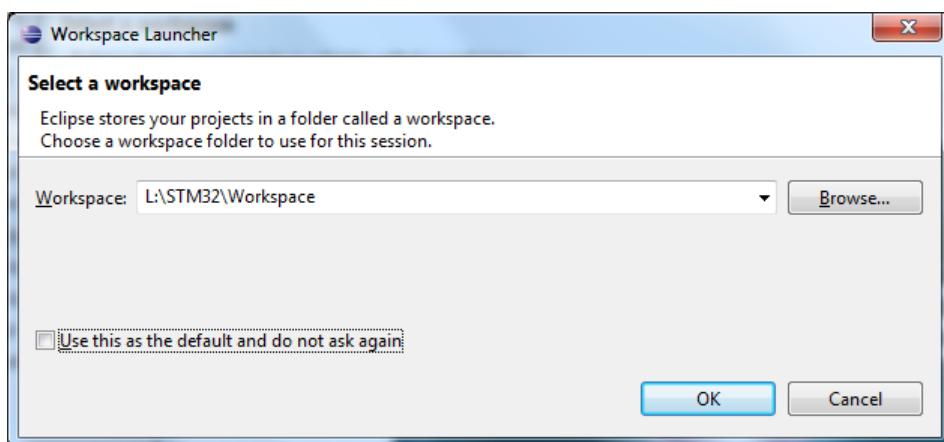


En finalitzar, dins de la carpeta haureu de tenir els següents directoris:

Doc	Manuals PDF i vincles d'Internet
Practica	Fitxers de la primera pràctica (Projecte preconfigurat)
Workspace	Directori de treball per Eclipse (Workspace preconfigurat)

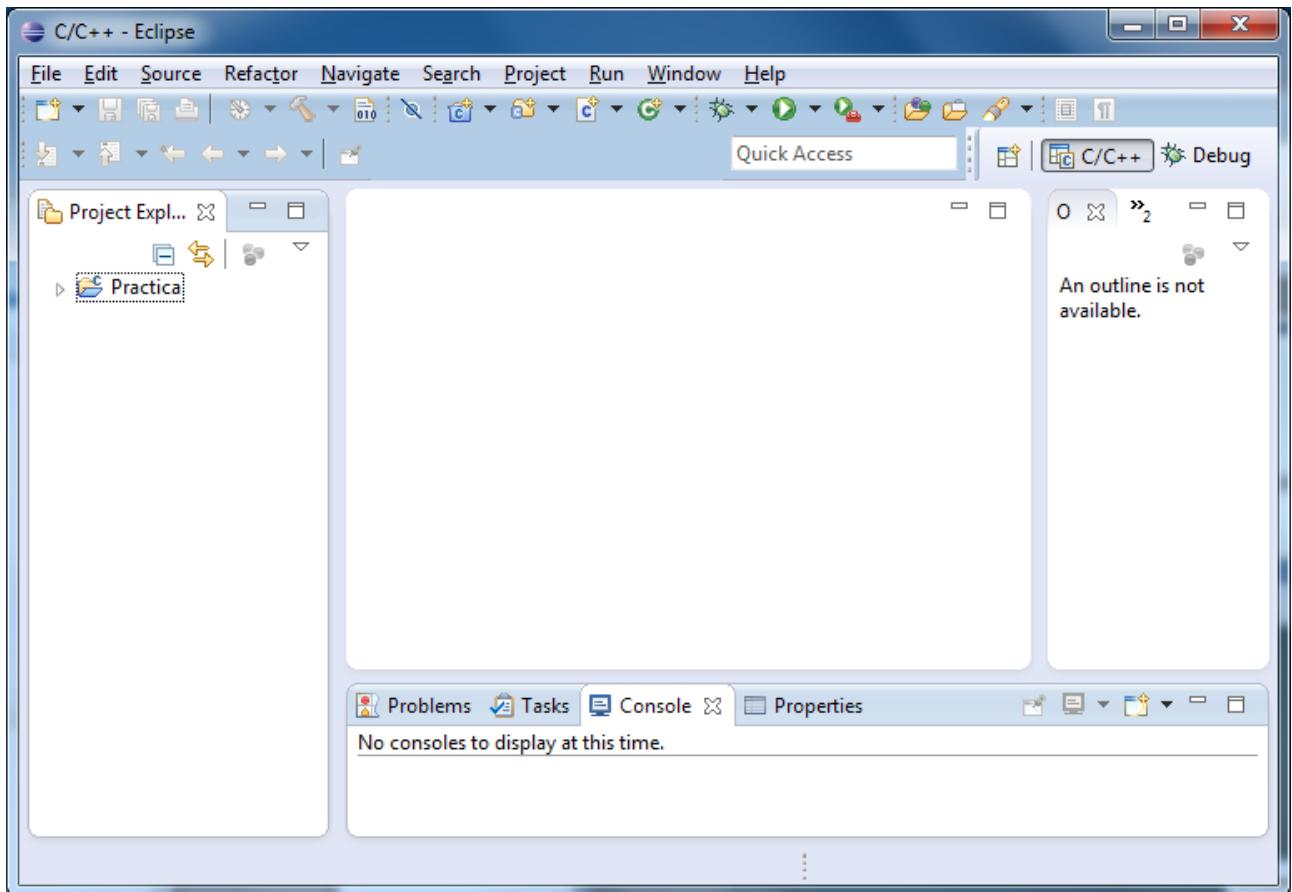
**És molt important que el directori STM32 pengidirectament de L:\ i que respecteu l'estructura de directoris que s'hi indica.**

A continuació engegueu Eclipse amb la icona **STM32 Eclipse** de l'escriptori. El primer que farà el programa es demanar un directori de treball (Workspace). Polseu sobre **Browse...** i indiqueu el vostre directori **Workspace**.



Després polseu **OK**.

Eclipse engegarà amb la següent pantalla:

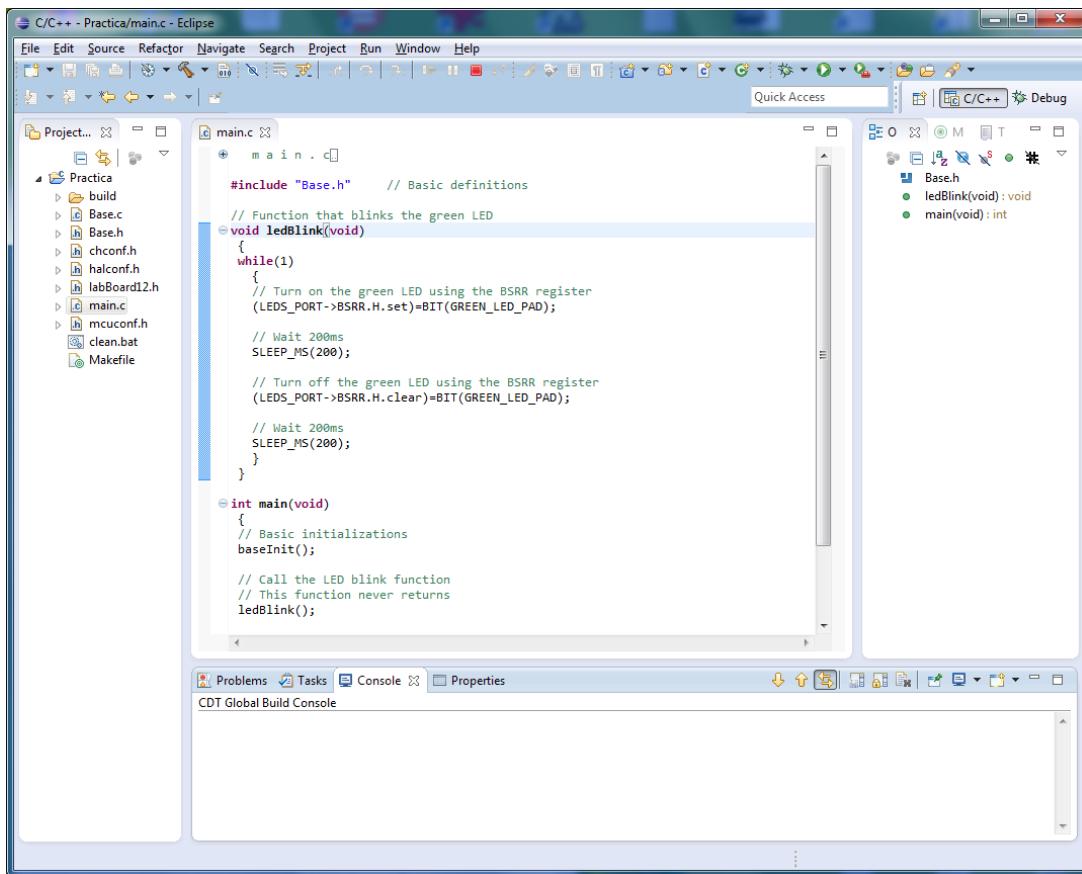


### P1.3 Compilació del projecte

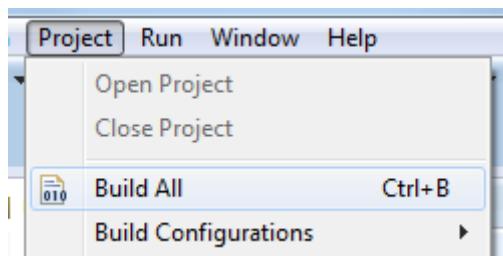
Si heu fet els passos correctament, ja teniu a Eclipse un projecte de nom "**Practica**".

Polsant sobre la petita icona que apareix a l'esquerra del dibuix de la carpeta es desplegaran els fitxers continguts dins el projecte.

Aquests fitxers els podeu examinar fent-hi doble clic al damunt, de manera que es vegin a la finestra central d'edició. A continuació es mostra un exemple després d'obrir el fitxer "**main.c**".



Per compilar el projecte podeu seleccionar al menú **Project → Build All**



Com a resultat hauria de sortir una finestra com la següent. Observeu que s'hi indica que la compilació ha estat correcta, a la consola, donat que acaba amb "*Build Finished*".

```

CDT Build Console [Practica]
mkdir -p build/obj
mkdir -p build/lst
Compiling Base.c
Compiling main.c
Chibios/RT is linked from precompiled objects
Linking build/ch.elf
Creating build/ch.hex
Creating build/ch.bin
Creating build/ch.dmp
Done

16:56:40 Build Finished (took 5s.260ms)

```

Si voleu estar segurs que la compilació ha acabat correctament podeu mirar si s'ha generat l'executable final **ch.elf** dins del subdirectorí **build** del vostre projecte.

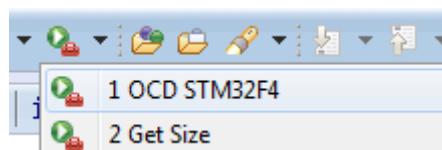
Com que aquest és un projecte basat en *Makefile*, de fet, no cal fer servir Eclipse per compilar-lo. Només cal que les eines de compilació es trobin al *path* del sistema i es podria compilar directament des d'una finestra de comandes.

#### **P1.4 Càrrega del projecte a la RAM de la placa**

Aquest projecte exemple que us donem només fa que el led verd de la placa STM32F4 *Discovery* s'encengui i s'apagui de manera intermitent. Per carregar el programa a la placa, primer ens assegurarem de que estigui connectada a un port USB de l'ordinador. Si hem fet el punt P1.1 correctament, l'LCD hauria de mostrar el missatge d'arrencada del test després de pulsar el botó **Reset**.

La depuració del projecte es fa mitjançant un servidor GDB. Aquest és un programa que està en marxa de manera permanent i que estableix la comunicació entre el PC i la placa de pràctiques. En principi aquest programa només s'ha d'ençegar una vegada cada cop que connectem la placa al PC.

Per arrencar el servidor GDB seleccioneu l'opció "GDB Server" o "OCD STM32F4" al menú de la icona amb forma de maleta que accedeix a les aplicacions externes.



Haurien de sortir uns missatges com els que es mostren a continuació.

```
GDB Server [Program] C:\STM32v2\openocd-0.8.0\bin\openocd-0.8.0.exe
Open On-Chip Debugger 0.8.0 (2014-04-28-08:39)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.sourceforge.net/doc/doxygen/bugs.html
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : This adapter doesn't support configurable speed
Info : STLINK v2 JTAG v14 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.894159
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

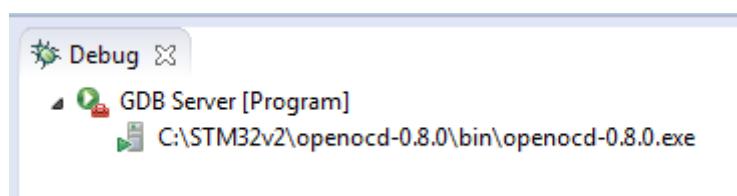
The screenshot shows the Eclipse IDE's Console view. The title bar says 'Console'. The main area displays the log output of the GDB server starting up. It includes the version information, license, bug report URL, adapter configuration, and target details for the STM32F4 chip.

Ja tenim en marxa la comunicació amb la placa. Ara només queda carregar-hi el programa.

A Eclipse tenim, per defecte, dues perspectives de treball, **C/C++** i **Debug**. Aquestes perspectives canviem la configuració de finestres (vistes) que tenim a la pantalla i són completament personalitzables. En general, per fer la depuració, canviarem a la perspectiva de **Debug**.



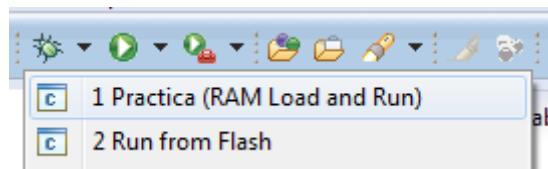
En canviar a la perspectiva de debug veurem que el servidor GDB funciona correctament:



Si el programa **openocd** no surt a la finestra de debug, llavors el servidor GDB no ha engegat correctament. També podem comprobar que **openocd** funciona correctament perquè el LED LD1 de la placa te colors verd/vermell intermitents. Això indica que hi ha comunicació al port USB.

Per depurar el nostre programa hem d'arrencar una nova sessió de debug:

Seleccioneu la opció **Practica (RAM Load and Run)** dins del menú de la icona de **debug** (forma de escarabat).



El *Makefile* del projecte està configurat per carregar el programa a la memòria RAM del microcontrolador en lloc de fer servir la memòria Flash. Amb això complim dos objectius: En primer lloc no desgastem la Flash que té un nombre limitat de cicles d'escriptura i en segon lloc mantenim el programa de test resident dins de la Flash.

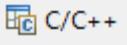
D'altra banda treballar a la RAM té alguns inconvenients:

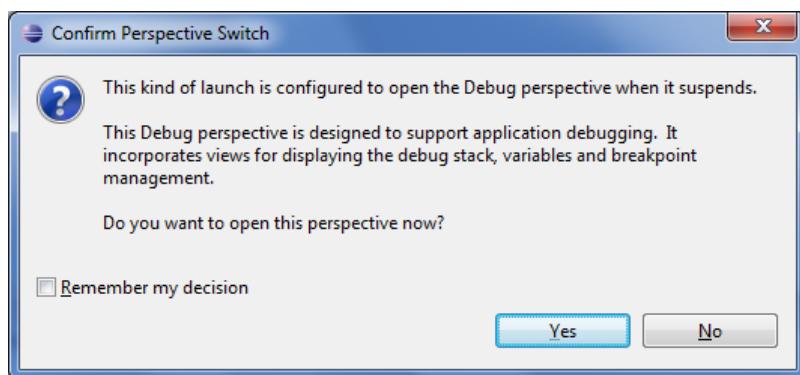
- Hi ha menys espai que a la Flash
- No té la mateixa temporització que la Flash
- El programa es perd quan s'apaga la placa

Si feu servir un projecte més complex que no hi càrga a la RAM o que calgui que sigui persistent quan apaguem l'alimentació, es pot compilar i executar dins la Flash modificant el Makefile. Parleu però amb el professor de pràctiques abans de fer aquest tipus de modificacions.

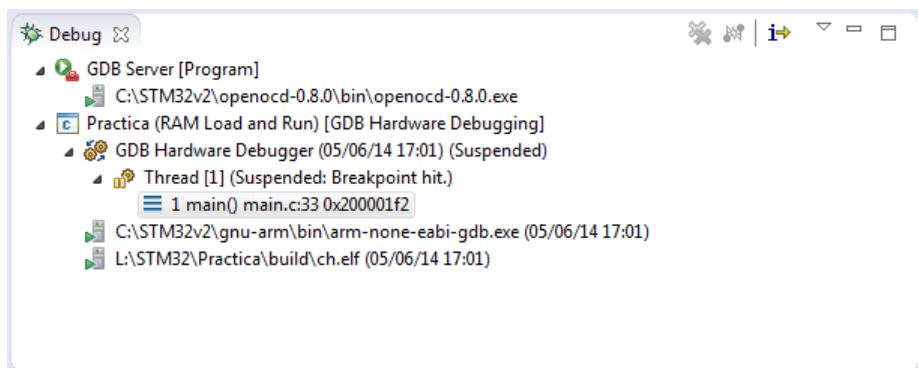
Si per algun motiu acabeu escrivint a la Flash, igualment haureu de fer servir la opció **Practica (RAM Load and Run)** tot i que carregareu a la Flash enllloc de a la RAM.

La opció **Run from Flash** només te interès si estem treballant amb la memòria flash i volem tornar a arrencar el programa amb el codi que ja hem carregat prèviament a la Flash. Donat que al laboratori treballarem amb la RAM, no cal que feu servir aquesta opció.

Si us trobàveu a la vista **C/C++**, sortirà un missatge com el següent. Seleccioneu *Remember my decision* si no voleu que torni a sortir i poleu **Yes**. Amb això canviem la perspectiva d'Eclipse de la perspectiva  **C/C++**, adequada per editar programes, a la perspectiva  **Debug**, adequada per carregar i depurar programes.



Dins de la perspectiva **Debug**, hauria de sortir una imatge com la que es mostra a continuació a Eclipse.



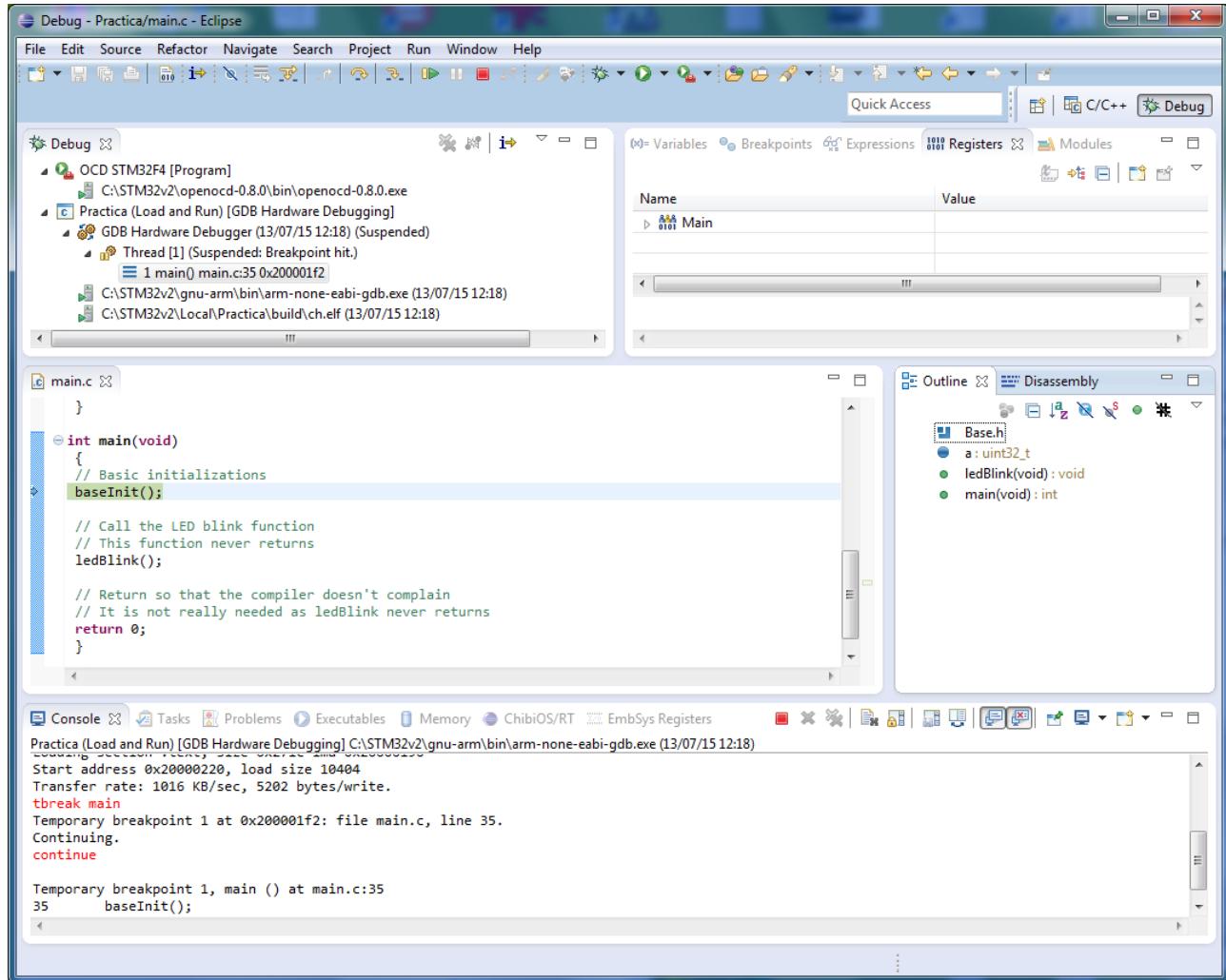
Ens trobem al programa principal en estat de parada al principi de main()

Si volem que es posi en marxa només cal que prenem el botó **Resume** .



En alguns cassos caldrà pulsar aquest botó dues vegades.

Si tot va bé el led verd de la placa hauria de començar a fer polsos. La pantalla LCD, com que no la fem servir hauria de mostrar l'últim missatge de test o res en absolut.



Per parar el programa podem pulsar el quadre vermell "Terminate"



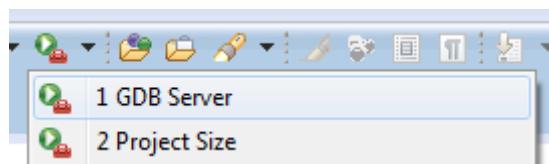
El led deixa de fer pampallugues i surt el missatge <terminated, exit value:0> gdb a la finestra de debug. Podeu eliminar el missatge pulsant sobre "terminated" i prenent a continuació la tecla Supr.

## P1.6 Depuració del projecte

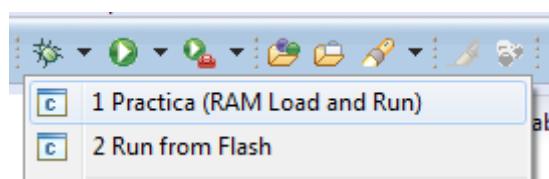
Amb Eclipse podem depurar el projecte, a més de carregar-lo a la placa.

Observeu que el servidor GDB és encara en marxa. Si el volguéssiu parar podeu polsar sobre **GDB Server [Program]** i polsar a continuació el quadre vermell  (Terminate). Per buidar la finestra podeu polsar la tecla **Supr** a continuació.

Per tornar a engegar el servidor GDB podeu polsar el triangle negre a la dreta de la icona que té una petita maleta vermella (External Tools) i seleccionar "GDB Server".



Si volem tornar a arrencar el programa només cal polsar el triangle negre a la dreta del escarabat (Debug) que hi sota el menú i escollir **Practica (RAM Load and Run)**. El programa torna a arrencar i para al principi de main().



En lloc de polsar *Resume* podem fer servir una de les altres comandes. Passeu el ratolí per sobre de les icones al costat de la de "*Resume*" i trobeu "*Step Into*"  i "*Step Over*" .



*Step Into*  permet executar el programa pas a pas entrant a les funcions quan es troben, *Step Over*  també permet executar pas a pas però sense entrar en les funcions. Per provar-ho, polseu *Step Over*  per saltar la instrucció **baseInit**, a continuació polseu *Step Into*  per entrar dins de **ledBlink**. Una vegada dins de **ledBlink**, aneu polsant *Step Over*  per veure com s'executa pas a pas i es va encenent i apagant el led. En qualsevol moment podeu tornar a polsar *Resume*  per que el programa continuï de manera autònoma.

Mentre corre el programa, en qualsevol moment, podeu polsar "*Suspend*" , per parar el programa. El punt de parada pot no trobar-se, però, a main() ni a ledBlink().

Polseu "*Terminate*"  per acabar l'execució.

A part de fer execució pas a pas podem veure el contingut de les variables mentre s'executa el programa i podem fer també parades a punts concrets del programa (*breakpoints*).

Per provar-ho escriviudins de **main.c** les dues línies indicades de manera que quedí com s'indica:

```
*****  
m a i n . c  
Practica 1  
*****  
  
#include "Base.h"      // Basic definitions  
  
volatile uint32_t i=0;  
  
// Function that blinks the green LED  
void ledBlink(void)  
{  
    while(1)  
    {  
        i++;  
        // Turn on the green LED using the BSRR register  
        (LEDS_PORT->BSRR.H.set)=BIT(GREEN_LED_PAD);  
  
        // Wait 200ms  
        SLEEP_MS(200);  
  
        // Turn off the green LED using the BSRR register  
        (LEDS_PORT->BSRR.H.clear)=BIT(GREEN_LED_PAD);  
  
        // Wait 200ms  
        SLEEP_MS(200);  
    }  
}  
  
int main(void)  
{  
    // Basic initializations  
    baseInit();  
  
    // Call the LED blink function  
    // This function never returns  
    ledBlink();  
  
    // Return so that the compiler doesn't complain  
    // It is not really needed as ledBlink never returns  
    return 0;  
}
```

Aquestes línies afegeixen una variable global "**i**" que s'incrementarà dins del bucle que hi ha a **ledBlink**. Observeu que la variable es declara **volatile**. Això ho fem per que si no, l'optimitzador que fa servir el compilador podria suprimir el codi que hem afegit ja que no fa res pràctic.

No cal sortir a la perspectiva "C/C++" per fer aquests canvis, però es pot fer si es vol.



Salvem el fitxer amb la icona del diskette i tornem a compilar amb **Project → Build All** o, més fàcilment amb **Ctrl+B**.

Després de compilar sense errors podem carregar el nou programa amb el triangle de la dreta de la icona de *Debug*. El programa, es pararà a l'inici de main().

Posarem ara un breakpoint a la línia que conté `i++;` per fer-ho polsarem el botó dret del ratolí a la zona ombrejada a la esquerra d'aquesta i seleccionarem **Toggle Breakpoint** amb el menú contextual. Hauria de sortir quelcom com el que es mostra a la figura.

```

volatile uint32_t i=0;

// Function that blinks the green LED
void ledBlink(void)
{
    while(1)
    {
        i++;
        // Turn on the green LED using the BSRR register
        (LEDS_PORT->BSRR.H.set)=BIT(GREEN_LED_PAD);

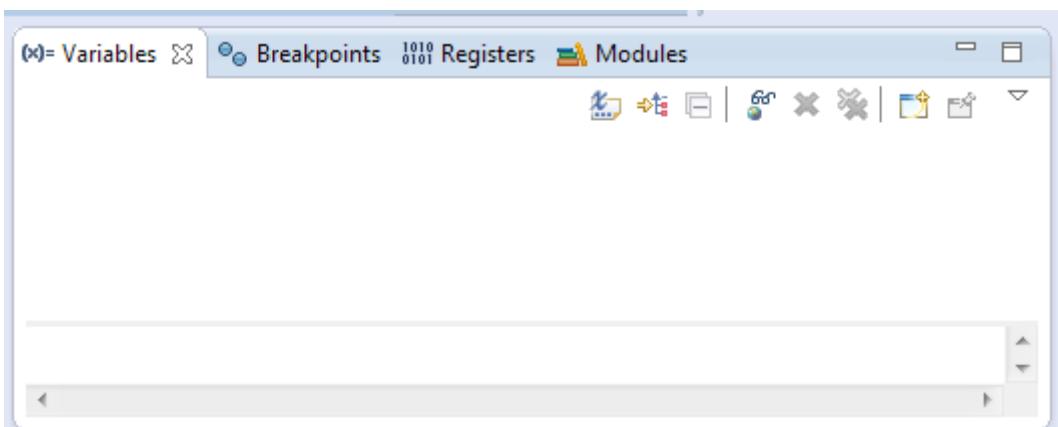
        // Wait 200ms
        SLEEP_MS(200);

        // Turn off the green LED using the BSRR register
    }
}

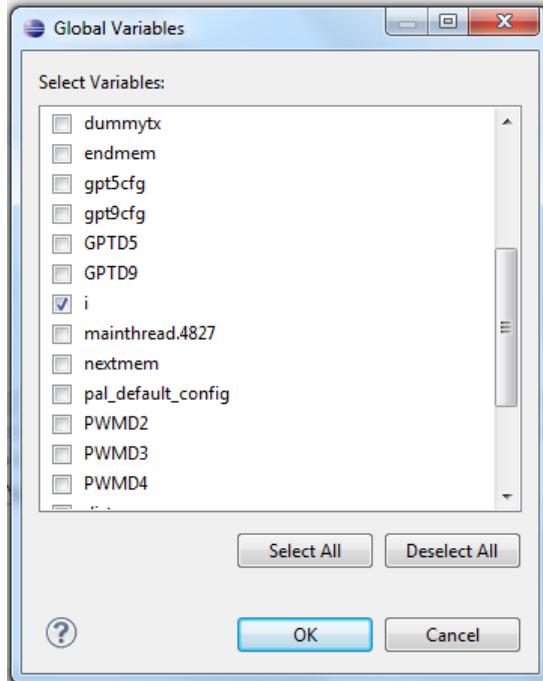
```

Observeu que el *breakpoint* es mostra amb un petit cercle de color blau. Pitgem ara el botó *Resume* (triangle verd). El programa continua l'execució fins que arriba al *breakpoint*.

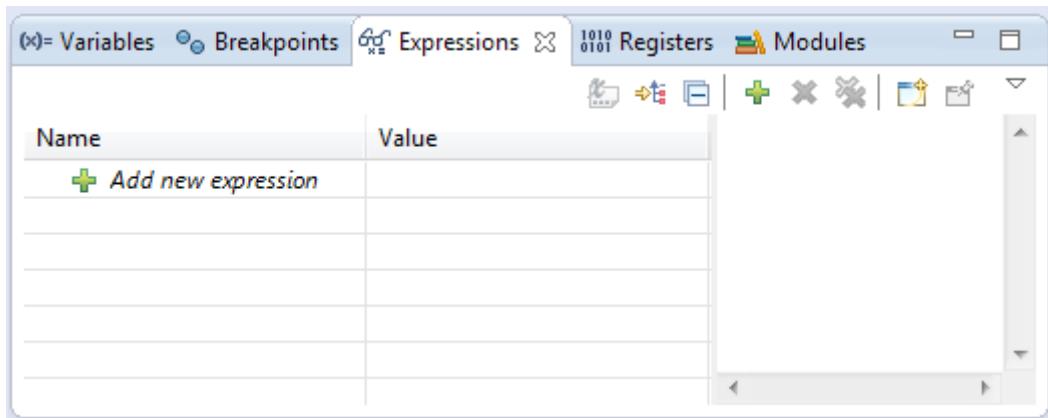
Podem observar la evolució de la variable i quan executem el programa al menys de dues maneres. La primera correspon a fer servir la finestra de variables globals



Feu clic sobre la icona que te unes ulleres i s'obrirà la següent finestra on podeu seleccionar les variables globals que vulgueu:



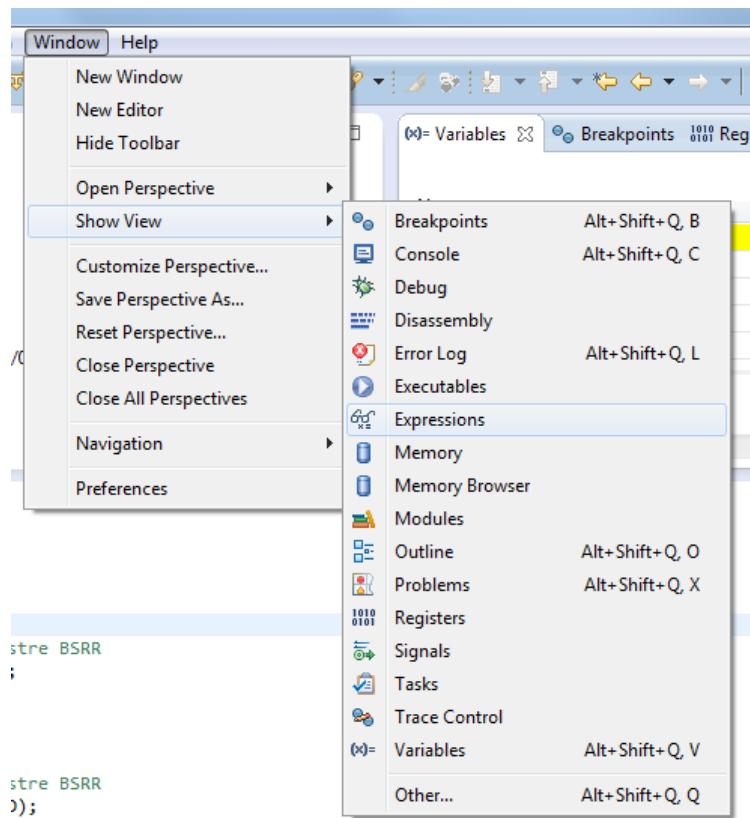
Addicionalment també podem fer servir la finestra d'expressions per veure el contingut de les variables o, de fet, de qualsevol expressió.



Es possible que aquesta finestra no es trobi a la vista de *Debug* tal i com s'indica a la figura anterior. Si aquest es el cas, podeu afegir aquesta finestra seleccionant el menú:

Window → Show View → Expressions

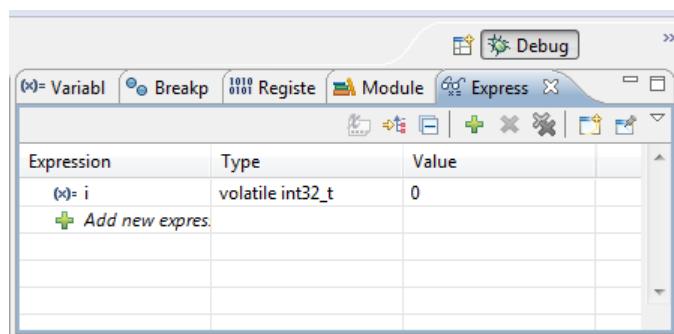
Tal i com s'indica a la figura següent:



Com podeu veure les perspectives de C/C++ i *Debug* son completament configurables. Podeu afegir o treure els elements que vulgueu de qualsevol de les dues perspectives.

Per poder veure com evoluciona el valor de la variable **i** a la finestra d'Expressions polsem sobre "+ Add new expression" i escrivim la variable **i**.

Trobarem el valor actual de la variable "i" just abans d'incrementar que es de zero.



Per cada vegada que es polsi el botó *Resume* , el bucle farà una volta i es pararà al *breakpoint*, de manera que podrem veure com es va incrementant el valor de i.

Després d'això podem tancar la sessió de *Debug* pitjant *Terminate* .

Els *breakpoints* es poden treure de la mateixa manera que s'hi posen. Només cal posar el ratolí a sobre i escolli **Toggle Breakpoint** amb el menú contextual del botó dret.

## P1.7 Descripció del funcionament del programa

Després de veure com funciona el procés de compilació i depuració, podem estudiar en detall el funcionament del programa. Per veure-ho amb mes comoditat podem fer servir la perspectiva C/C++.

El programa consta de 7 fitxers font: Makefile, mcuconf.h, halconf.h, chconf.h, Base.c, Base.h i main.c. Descriurem aquests fitxers un a un.

### Makefile

Aquest fixter indica quins fitxers fonts es fan servir i com s'han de compilar. Básicament aquest fixter descriu tots els passos necessaris per generar el codi fina a partir dels fitxers font.

Descriure el contingut i funcionament d'un fitxer makefile excedeix els objectius d'aquest laboratori.

La part més important, en el nostre cas, es la que es mostra a continuació:

```
##### PROJECT SOURCE FILES #####
# The source .c files that make the project can be added
# between Base.c and main.c
# The character "\" at the end of all but the last line is important
PCSRC = Base.c \
        main.c
```

PCSRC es el conjunt de fitxers "C" del projecte que s'han de compilar. De moment conté només els fitxers **Base.c** i **main.c** que es troben al nostre projecte. Si volguéssim afegir mes fitxers de codi ho faríem afegint-los entre **Base.c** i **main.c** sense oblidar que les línies, llevat de l'última que conté **main.c**, han d'acabar amb el caràcter "\" per indicar que la definició continua a la línia següent.

El fitxer *Makefile* conté referències a altres fitxers font con son els del sistema operatiu ChibiOS/RT, dins la variable BCSRC, però aquests no els hem de tocar:

```
# C sources that can be compiled in ARM or THUMB mode depending on the global
# setting.
BCSRC = $(PORTSRC) \
        $(KERNSRC) \
        $(HALSRC) \
        $(PLATFORMSRC) \
        $(BOARDSRC)
```

El *Makefile* conté també tres variables especials que controlen com es fa la compilació del projecte:

**REBUILD** = no

Aquesta variable indica que només s'han de compilar els fitxers propis del nostre projecte indicats a PCSR com son Base.c i main.c. Els fitxers indicats a BCSRC en lloc de ser compilats s'enllaçaran al fixter final fent servir els objectes precompilats que es troben a C:\STM32v2\Objects.

No heu de canviar aquesta variable per fer les pràctiques. Aquesta variable però, pot interesar de ser canviada si, per altres projectes aliens a l'assignatura, feu canvis de configuració del sistema operatiu o si només feu servir compilació local.

#### **RAM\_COMPILE = yes**

Aquesta variable indica que els fixers font s'han d'executara a la memòria RAM en lloc de la memòria Flash. Aquesta opció permet minimitzar el desgast de la memòria Flash ja que aquesta es pot escriure només un nombre finit de vegades.

Aquesta opció limita la mida del codi a 64KB i les dades a 48KB. També fa que el programa es perdi quan es treu l'alimentació.

Aquestes limitacions no son problema per fer les pràctiques. Si feu servir la placa en altres projectes pot ser interessant modificar aquesta variable si el codi o les dades superen el límits anteriors o si cal que el programa sigui persistent a la memòria del microcontrolador.

#### **USE\_FPU= yes**

Per defecte es configura el compilador per fer servir acceleració hardware de càlculs en coma flotant. Això accelera molt els càlculs però augmenta les necessitats de pila en canvis de contexte.

Finalment, s'ha de comentar que el *Makefile* conté per defecte aquestes línies:

```
ifeq ($(USE_OPT),)
#USE_OPT = -O2 -ggdb -fomit-frame-pointer -falign-functions=16
USE_OPT = -O0 -ggdb -fomit-frame-pointer -falign-functions=16
endif
```

Aquestes fan que la compilació es faci sense optimitzacions "**-O0**". Aquesta opció es la més adequada per poder fer depuració pas a pas sense problemes. Una vegada completament depurat el programa, i si ens cal optimitzar la mida de codi o la seva velocitat, es pot descomentar la primera línia i comentar la segona de manera que tingui efecte la opció "**-O2**" que activa les optimitzacions.

### **mcuconf.h, halconf.h i chconf.h**

Aquests tres fitxers son els fitxers de configuració del microcontrolador STM32F407, la capa d'abstracció del hardware (HAL) i del sistema operatiu (ChibiOS) respectivament. Podeu examinar el seu contingut però, de moment, millor no tocar-ho.

Si toqueu algun contingut d'aquests fitxers haureu de canvira la variable *REBUILD* per la que te la opció "*yes*" ja que els objectes precompilats només son vàlids pels valors per defecte d'aquests fitxers.

## **Base.c i Base.h**

Aquest dos fitxers subministren alguns elements de suport pel nostre programa. **Base.c** conté tres funcions: **gptDelayUs** i **gptDelayUs2** que es fan servir per fer retards de durada controlada i **baseinit** que es la funció que hem de cridar en començar **main** per posar en marxa la placa correctament. **Base.h** conté els prototips d'aquestes funcions i un conjunt addicional d'elements d'ajuda per desenvolupar el nostre programa:

- Crides als fitxers include "ch.h", "hal.h" i "labBoard12.h"
- Macro BIT(n) que genera el valor  $2^n$
- Macros BITxx que corresponen a tots els valor des de  $2^0$  a  $2^{31}$
- Macros DISABLE i ENABLE para desactivar/activar el sistema operatiu i les interrupcions
- Macros **DELAY\_US(x)** i **DELAY\_US2(x)** que fan un retard de x microsegons.
- Macro **SLEEP\_MS(x)** que adorm el processador durant x milisegons.

Les dues funcions **gptDelayUs** i **gptDelayUs2** associades respectivament a **DELAY\_US(x)** i **DELAY\_US2(x)** tenen un funcionament idèntic pero fan servir comptadors independents. Aixó vol dir que es poden fer servir a dos fils d'execució diferents.

## **labBoard12.h**

Aquest fitxer es un **board file** de la versió 1.2 de la placa de pràctiques. Conté la descripció dels elements de hardware de la placa de pràctiques i els recursos del microcontrolador que fan servir. En concret s'indiquen els recursos dels següents perifèrics:

Per la placa STM32F4 Discovery:

- 4 LEDs d'usuari
- Botó d'usuari
- Acceleròmetre de la placa LIS302DL
- Micròfon de la placa MP45DT02
- DAC d'audio de la placa CS43L22

Pels components de la placa de pràctiques fora de la placa Discovery:

- LCD alfanumèric
- Teclat
- Potenciòmetre
- Encoder

Observeu que per molts pins hi ha dos tipus de definicions:

PAD : Es el nombre del pin associat a un port. Aquest valor pot anar de 0 a 15

BIT : Correspon a  $2^n$  on n es el nombre del pin.

Per tant, per exemple, RED\_LED\_BIT es equivalent a  $2^{\text{RED\_LED\_PAD}}$  que, fent servir l'operador de desplaçament a la esquerra es ( $1 << \text{RED\_LED\_PAD}$ ). Fent servir la macro BIT serà també BIT(RED\_LED\_PAD).

Les definicions incloses al fitxer *labBoard12.h* s'han de fer servir dintre del vostre codi. Per exemple, per encendre el LED vermell, en lloc del següent codi:

```
(GPIOD->BSRR.H.set)=BIT14;
```

Farem servir el el codi:

```
(LEDS_PORT->BSRR.H.set)=BIT(RED_LED_PAD);
```

O bé:

```
(LEDS_PORT->BSRR.H.set)=RED_LED_BIT;
```

Aquesta norma es molt important per fer el vostre codi el més independent possible de la placa que feu servir. D'aquesta manera, qualsevol canvi a la placa es pot codificar amb canvis al fitxer *.h* sense canviar el vostre codi al fitxer *.c*. De fet, si tinguessim dues plaques de pràctiques amb els mateixos perifèrics però amb diferents connexions, podríem canviar d'una placa a l'altra sense més que canviar el fitxer de placa *.h* que fem servir.

En els dissenys embedded es habitual separar les definicions concretes del elements de hardware del codi fent servir fitxers com *labBoard12.h*. A més, es molt important quan s'escriu codi que cada element únic estigui definit només una vegada. Per exemple, el LED vermell es troba a la placa a la línia 14 del port D. El nostre codi pot fer servir aquest LED a molts llocs. Imaginem que a cada lloc fem servir els valors 14 i D. Si un dia cal canviar el programa per funcionar a un altre placa, haurem de trobar tots el llocs del codi on es troben aquestes referències per canviar-les. Això fa que sigui molt probable que ens oblidem alguna. Aixó no aplica només al contingut de *labBoard12.h*. Aplica a qualsevol constant que feu servir. En general, qualsevol constant que feu servir més d'una vegada al vostre codi (o que poguen haver de canviar en el futur) convé que estigui definida amb una constant **#define** dins d'un fitxer de capçalera *.h* amb el seu corresponent comentari explicatiu.

### **clean.bat**

Aquest fitxer executable esborra els directoris **.dep** i **build** del nostre projecte.

Amb això s'eliminen tots el fixters objecte i executables. Donat que aquests fitxers es generen automàticament aquest executable permet compactar la informació del projecte y/o garantir que no hi ha cap fitxer objecte compilat indegudament.

## main.c:

Aquest es el nostre programa principal pel programa que fa els polsos al led verd.

```
*****  
m a i n . c  
Practica 1  
*****  
  
#include "Base.h"      // Basic definitions  
  
volatile uint32_t i=0;  
  
// Function that blinks the green LED  
void ledBlink(void)  
{  
    while(1)  
    {  
        i++;  
        // Turn on the green LED using the BSRR register  
        (LEDS_PORT->BSRR.H.set)=BIT(GREEN_LED_PAD);  
  
        // Wait 200ms  
        SLEEP_MS(200);  
  
        // Turn off the green LED using the BSRR register  
        (LEDS_PORT->BSRR.H.clear)=BIT(GREEN_LED_PAD);  
  
        // Wait 200ms  
        SLEEP_MS(200);  
    }  
}  
  
int main(void)  
{  
    // Basic initializations  
    baseInit();  
  
    // Call the LED blink function  
    // This function never returns  
    ledBlink();  
  
    // Return so that the compiler doesn't complain  
    // It is not really needed as ledBlink never returns  
    return 0;  
}
```

Tots els nostres programes han de començar amb una línia:

**#include "Base.h"**

aquesta línia fa que s'incloguin tots els fitxers .h que permeten gestionar la placa.

A continuació ve la funció **ledBlink**. Aquesta conté tota la funcionalitat del nostre primer programa. La segueix la funció **main** que es el punt d'entrada del programa que carreguem a la nostra placa. La funció **main** sempre ha de començar cridant **baseInit()** per tal que es posi en marxa correctament la placa de pràctiques.

Seguidament es crida a la nostre funció **ledBlink**. Aquesta funció té un bucle infinit en el que, de manera repetitiva, s'encén el led verd, s'espera 200ms, s'apaga el led verd, i es torna a esperar 200ms.

La funció no retorna mai, per tant, la resta de **main** no s'executarà mai i **main** no retornarà.

La gestió del led requereix una explicació molt més detallada.

El microcontrolador STM32F407V disposa de 82 pins de GPIO (General-Purpose Input/Output) tal i com s'indica a la pàgina 12 del fitxer "[STM32F407 Datasheet.pdf](#)".

Aquests pins estan agrupats en ports de 16 bits cadascun. En concret tenim els 5 ports A, B, C, D, E amb 16 bits cadascun, i un 6è port H amb 2 pins. Normalment designem una línia de GPIO amb "P", la lletra del port, i el nombre de línia (o PAD). Així, per exemple, **PC4** correspon al PAD 4 del port C.

**Table 2. STM32F405xx and STM32F407xx: features and peripheral counts**

Peripherals	STM32F405RG	STM32F405VG	STM32F405ZG	STM32F407Vx	STM32F407Zx	STM32F407Ix
Flash memory in Kbytes	1024			512	1024	512
SRAM in Kbytes	System			192(112+16+64)		
	Backup			4		
FSMC memory controller	No			Yes		
Ethernet		No			Yes	
Timers	General-purpose			10		
	Advanced-control			2		
	Basic			2		
Random number generator				Yes		
Communication interfaces	SPI / I <sup>2</sup> S			3/2 (full duplex)		
	I <sup>2</sup> C			3		
	USART/UART			4/2		
	USB OTG FS	No			Yes	
	USB OTG HS	Yes			Yes	
	CAN			2		
Camera interface		No			Yes	
GPIOs	51	82	114	82	114	140
12-bit ADC				3		
	Number of channels	16	16	24	16	24
12-bit DAC				Yes		
	Number of channels			2		
Maximum CPU frequency				168 MHz		
Operating voltage				1.8 to 3.6 V <sup>(1)</sup>		

La descripció detallada de la programació dels ports de GPIO es troba a partir de la pàgina 136 del fitxer "[STM32F4 Reference.pdf](#)". A continuació farem un breu resum del que cal per fer aquesta primera pràctica.

El control de cada un dels 6 ports es fa amb un conjunt de 10 registres. Per accedir a aquests registres fem servir un punter associat a cada port. Així GPIOA es el punter associat al port A, GPIOB al port B i així successivament.

En el nostre model de programació, cada punter apunta a una estructura de tipus **GPIO\_TypeDef** que conté els següents elements:

```
typedef struct {
    volatile uint32_t      MODER;
    volatile uint32_t      OTYPER;
    volatile uint32_t      OSPEEDR;
    volatile uint32_t      PUPDR;
    volatile uint32_t      IDR;
    volatile uint32_t      ODR;
    volatile union {
        uint32_t          W;
        struct {
            uint16_t        set;
            uint16_t        clear;
        } H;
    } BSRR;
    volatile uint32_t      LCKR;
    volatile uint32_t      AFRL;
    volatile uint32_t      AFRH;
} GPIO_TypeDef;
```

Aquesta estructura ens permet accedir als 10 registres de cada port de GPIO. La definició de l'estructura es troba al fitxer **pal\_lld.h** que es carrega automàticament en fer **#include "Base.h"** i que es troba al directori: *ChibiOS\_2.6.2\os\hal\platforms\STM32\GPIOv2*

A mode de exemple, per accedir al registre ODR del port GPIO D farem servir la referència "**GPIOD->ODR**".

El registres associats a **GPIO\_TypeDef** permeten configurar i accedir als ports en lectura i escriptura. La funció **baseInit** s'encarrega de manera automàtica de programar les línies dels leds com a sortida i el pulsador d'usuari com a entrada. La definició de les línies usades es troba, com hem dit abans, a **labBoard12.h** on podem trobar que el GPIO dels leds es el port D (GPIOD) i que els bits son el 12, 13, 14 i 15 pels colors verd, taronja, vermell i blau respectivament. De la mateixa manera s'indica que pulsador d'usuari es troba a PA0 (GPIOA bit 0).

```
/* Four active high LEDs on board */
/* Hal initialization in baseInit() configures the */
/* LEDs in output push-pull mode */

#define LEDS_PORT      GPIOD // LEDs are on port D

#define GREEN_LED_PAD 12 // Green LED Pad (Left)
#define ORANGE_LED_PAD 13 // Orange LED Pad (Up)
#define RED_LED_PAD    14 // Red LED Pad (Right)
#define BLUE_LED_PAD   15 // Blue LED Pad (Down)

#define GREEN_LED_BIT  (1<<12) // Green LED Bit (Left)
#define ORANGE_LED_BIT (1<<13) // Orange LED Bit (Up)
#define RED_LED_BIT    (1<<14) // Red LED Bit (Right)
#define BLUE_LED_BIT   (1<<15) // Blue LED Bit (Down)

/* The board includes one user button */
/* There is a pull-down resistor so it is active at high level */
/* Hal initialization configures the button in input mode */

#define BUTTON_PORT     GPIOA // Button is on Port A
#define BUTTON_PAD      0    // Button Pad
#define BUTTON_BIT       1    // Button Bit
```

Part del contingut de Base.h

Per escriure a un port de sortida hem de fer servir el registre definit com **ODR** (Output Data Register) a **GPIO\_TypeDef**. Per llegir un port d'entrada hem de fer servir el registre definit com a **IDR** (Input Data Register) a **GPIO\_TypeDef**. En aquesta pràctica, però, encara no es fa falta fer servir el registre **IDR**.

#### 6.4.6 GPIO port output data register (GPIOx\_ODR) (x = A..I)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy[15:0]**: Port output data (y = 0..15)

These bits can be read and written by software.

*Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx\_BSRR register (x = A..I).*

[STM32F4 Reference.pdf](#)      Pàgina 150

El **Output Data Register (ODR)** es un registre de 32 bits del que en farem servir només els 16 bits mes baixos. Els 16 bits alts no s'han de tocar. Cada bit de la part baixa del registre correspon a una línia de sortida del port.

Com que el led verd es troba a la línia 12 del port D podríem encendre el led, que es actiu alt, amb la instrucció:

`GPIOD->ODR=BIT12;`

El problema es que el port D conté altres línies i, amb la instrucció anterior posaríem totes les línies que no son la 12 a 0. Per canviar només la línia 12 podem fer servir la funció OR booleana:

`GPIOD->ODR= (GPIOD->ODR) | BIT12;`

D'aquesta manera es fixa el bit 12, però els altres bits romanen iguals.

Com que el port D està compartit entre diversos perifèrics la instrucció anterior té el problema de que no es atòmica (d'un sol pas). El compilador la divideix en la seqüència:

```
temporal = GPIOD->ODR;
temporal = temporal | BIT12;
GPIOD->ODR = temporal;
```

Si després d'efectuar-se la primera instrucció de la seqüència, tenim la mala sort que es genera una interrupció i la RSI (Rutina de servei d'interrupció) modifica GPIOD de manera asíncrona, quan continuï la seqüència, ho farà amb informació errònia del contingut de GPIOD. D'aquesta manera, al final, es desfarà la modificació que s'havia intentat fer dins de la RSI. Una solució al problema es desactivar les interrupcions mentre s'accedeix a GPIOD de manera que la seqüència no s'interrompi. Això es faria amb la següent seqüència d'instruccions:

```
DISABLE;
GPIOD->ODR= (GPIOD->ODR)|BIT12;
ENABLE;
```

Aquesta solució funciona però no es òptima per que les crides a DISABLE i ENABLE incrementen el temps d'execució. Es per això que els ports GPIO disposen, en mode de sortida, del registre **BSRR** (Bit Set and Reset Register) que simplifica canviar a "0" o a "1" bits concrets del registre **ODR**.

La necessitat d'atomicitat depén del vostre programa. Al exemple d'aquesta pràctica el port D on es troben els LEDs no s'accedeix mai de manera asíncrona i, per tant, no cal en rigor fer servir **BSRR**. Es la vostra responsabilitat com a programadors fer servir instruccions atòmiques quan cal. En tot cas, mai no fa mal fer servir instruccions atòmiques quan no cal. El problema es **no fer-les servir** quan cal.

#### 6.4.7 GPIO port bit set/reset register (GPIOx\_BSRR) (x = A..I)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy**: Port x reset bit y (y = 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit  
1: Resets the corresponding ODRx bit

*Note: If both BSx and BRx are set, BSx has priority.*

Bits 15:0 **BSy**: Port x set bit y (y= 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit  
1: Sets the corresponding ODRx bit

El registre **BSRR** es un registre hardware especial de 32 bits. Un "1" als primers 16 bits del registre permet activar un dels bits del port, així, per exemple:

```
GPIOD->BSRR.W = BIT12;
```

es equivalent a:

```
GPIOD->ODR= (GPIOD->ODR)|BIT12;
```

però es una instrucció atòmica (d'un sol pas). Observeu que fem un accés de 32 bits a BSRR, per això l'element de l'estructura es **BSRR.W**, la "W" vol dir **Word Acces** o 32 bits.

De la mateixa manera, als 16 bits mes alts del registre **BSRR** un "1" permet desactivar un bit del port, així, per exemple:

```
GPIOD->BSRR.W = BIT28;
```

es equivalent a:

```
GPIOD->ODR= (GPIOD->ODR)&(~BIT12);
```

i, per tant, esborra el bit 12 del port D, ja que  $28-16 = 12$  (meitat alta de BSRR) + 12.

Escriure "0" a qualsevol bit de BSRR no modifica el port en absolut.

Per simplificar l'ús del registre BSRR, s'ha fet una definició alternativa on, els primers 16 bits de BSRR, que permeten activar una línia, corresponen a **BSRR.H.set** i els últims 16 bits de BSRR que permeten desactivar una línia, corresponen a **BSRR.H.clear**. Observeu que "H" vol dir "*Half Word*" (16 bits).

Així:

<code>PORTx-&gt;BSRR.H.set = BITn;</code>	Posa a "1" el bit n del port x.
<code>PORTx-&gt;BSRR.H.clear= BITn;</code>	Posa a "0" el bit n del port x.

On x es un port A...E, H i n es un bit 0...15

Si el valor de n està definit amb una variable o una constant definida amb una instrucció **#define**, en lloc de BITn, podem fer servir la macro BIT(n) que correspon a un desplaçament lògic a l'esquerra.

$$BIT(n) = (1 << n) = 2^n$$

Donat que hem definit LEDS\_PORT com GPIOD i GREEN\_LED\_PAD com 12, la instrucció:

```
(LEDS_PORT->BSRR.H.set)=BIT(GREEN_LED_PAD);
```

Activa el led verd que es troba a PD12, mentre que la instrucció:

```
(LEDS_PORT->BSRR.H.clear)=BIT(GREEN_LED_PAD);
```

Desactiva el led verd que es troba a PD12.

Alternativament també podem fer servir:

```
(LEDS_PORT->BSRR.H.set)= GREEN_LED_BIT;  
(LEDS_PORT->BSRR.H.clear)= GREEN_LED_BIT;
```

## **P1.8Modificació del programa**

Una vegada analitzat el funcionament del programa demanem que el modifiqueu de manera que els quatre leds es vagin encenent seguint la seqüència de manera repetitiva:

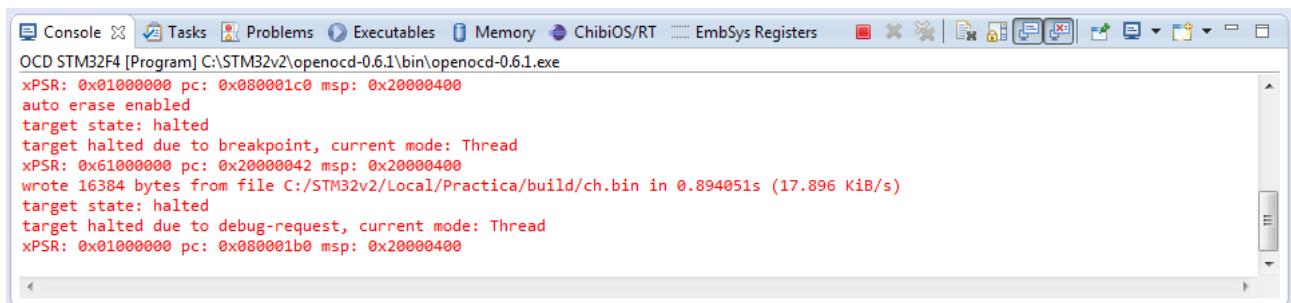
- Led Verd encès, resta apagats
- Led Taronja encès, resta apagats
- Led Vermell encès, resta apagats
- Led Blau encès, resta apagats

El temps que ha de estar encès cada led es 0.5s cada vegada que li toca.

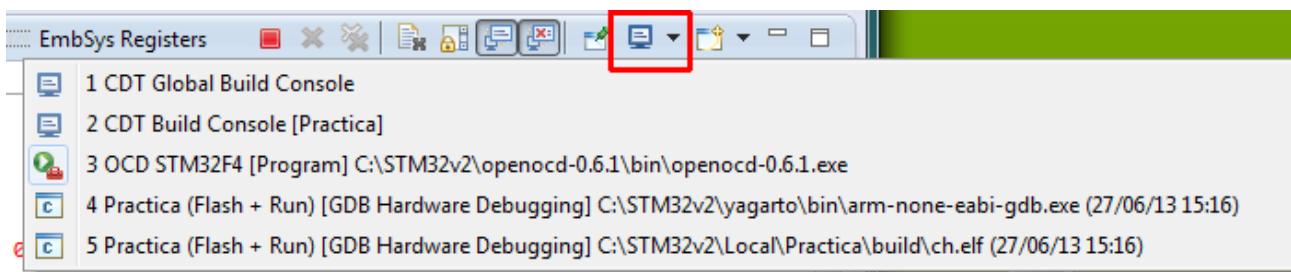
- Per fer-ho, crearem una nova funció **void ledSequence(void)** i farem que es cridi des de **main** en lloc de cridar **ledBlink**. No cal que esborreu la funció ledBlink, no molesta si no es fa servir.

## P1.9Eines adicionals de depuració

Ala part baixa de la pantalla, dins de la perspectiva de *Debug*, podeu trobar diferents eines. La primera d'elles és la consola.



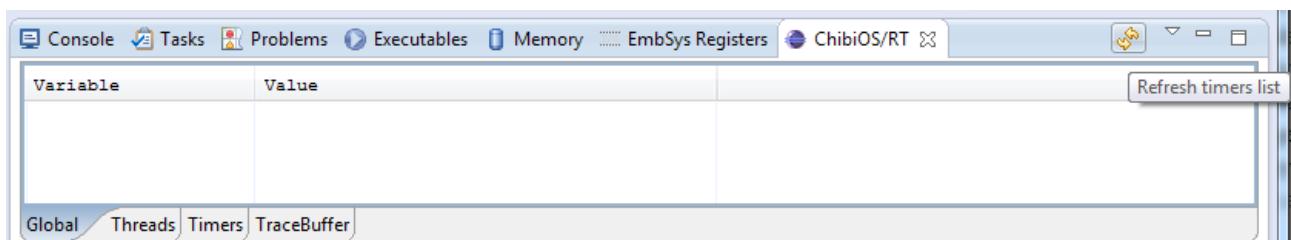
La consola permet observar la resposta en forma de text dels diferents programes que fem servir. Cada programa pot tenir la seva pròpia consola. Podem fer servir la icona de la consola per canviar entre les consoles dels diferents programes:



La pestanya **Problems** la fem servir normalment a la perspectiva **C/C++** per corregir els errors de compilació.

La pestanya **Memory** permet veure el contingut de la memòria del microprocesador, però no la farem servir gaire en aquestes pràctiques.

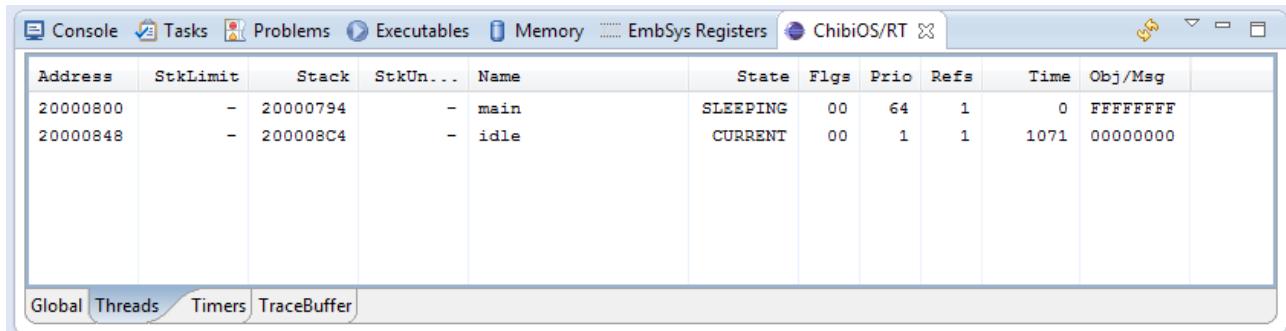
La pestanya **Chibios/RT** correspon a un plug-in associat al sistema operatiu en temps real que fem servir a les pràctiques. Aquest sistema el fem servir, de moment, només per arrencar la placa i controlar algunes de les funcions de retards.



Per actualitzar el contingut de la pestanya, s'ha de parar el programa amb "Suspend"  i polsar la icona "Refresh".

Quan fem les pràctiques R1 i R2 ens serà útil aquesta finestra per veure els fils de programa actius.

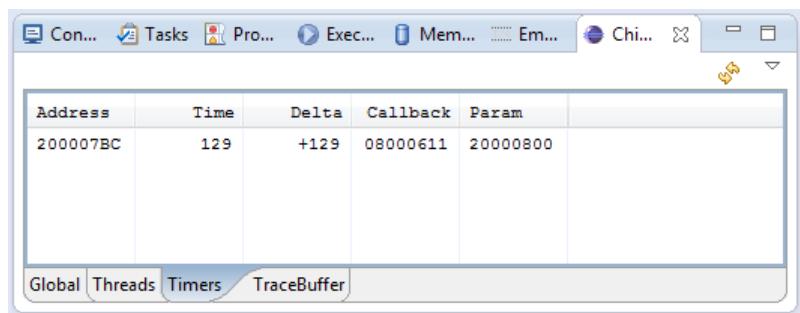
En el programa actual, únicament tenim dos fils, el principal en que corre **main** i l'**idle** que es posa en marxa quan cap procés pot córrer.



Address	StkLimit	Stack	StkUn...	Name	State	Flgs	Prio	Refs	Time	Obj/Msg
20000800	-	20000794	-	main	SLEEPING	00	64	1	0	FFFFFFFF
20000848	-	200008C4	-	idle	CURRENT	00	1	1	1071	00000000

Global Threads Timers TraceBuffer

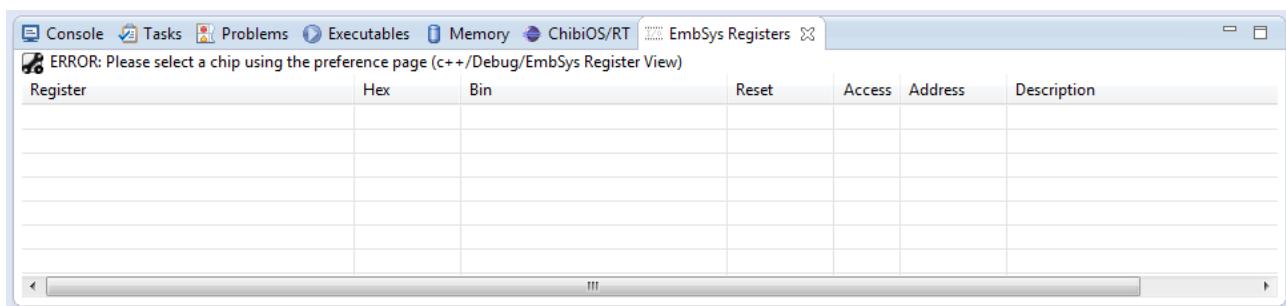
Per fer feines de temporització, **base.c** defineix dues funcions **gptDelayUs** i **gptDelayUs2** que fan servir temporitzadors del sistema operatiu. Aquests es pot veure a la pestanya Timers.



Address	Time	Delta	Callback	Param
200007BC	129	+129	08000611	20000800

Global Threads Timers TraceBuffer

La pestanya **EmbSys Registers** és un altre plug-in d'Eclipse que ens permet accedir al contingut dels registres hardware del microcontrolador.



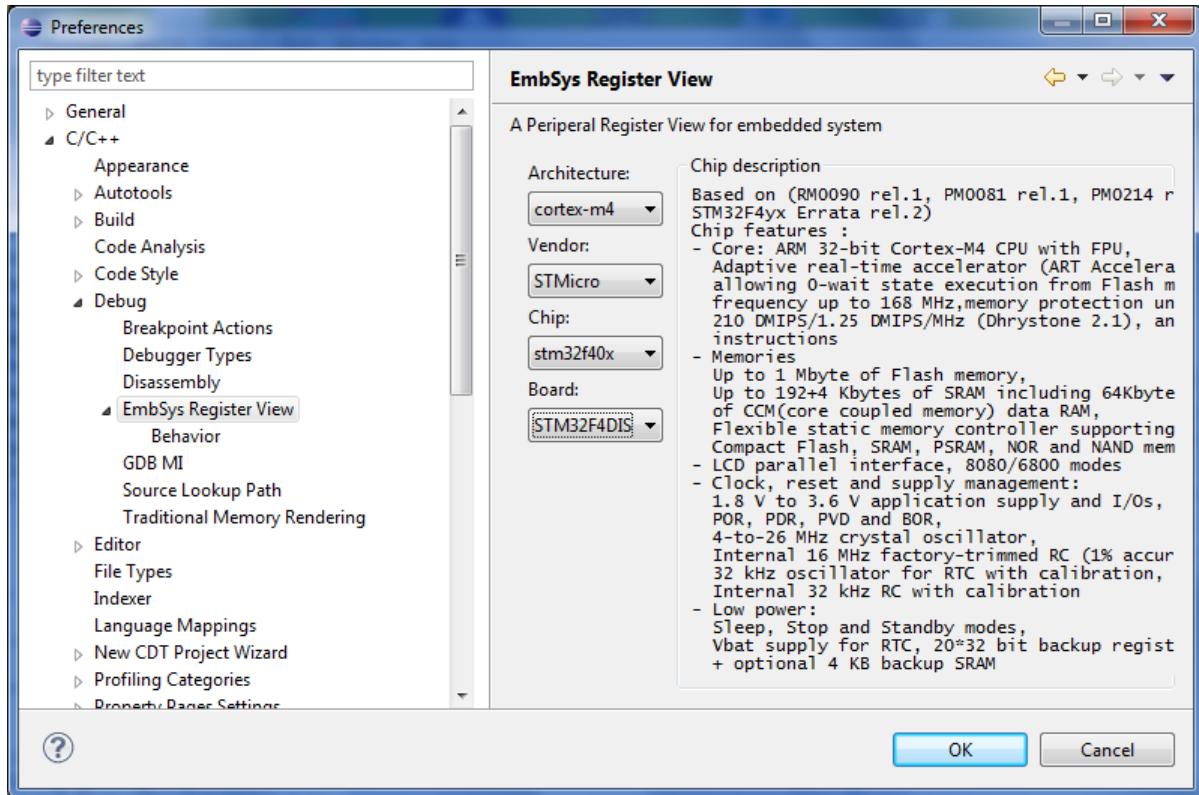
Register	Hex	Bin	Reset	Access	Address	Description

Si us surt la finestra anterior, vol dir que s'ha de configurar el plug-in pel processador que fem servir. Si no, podeu saltar el que s'indica a continuació.

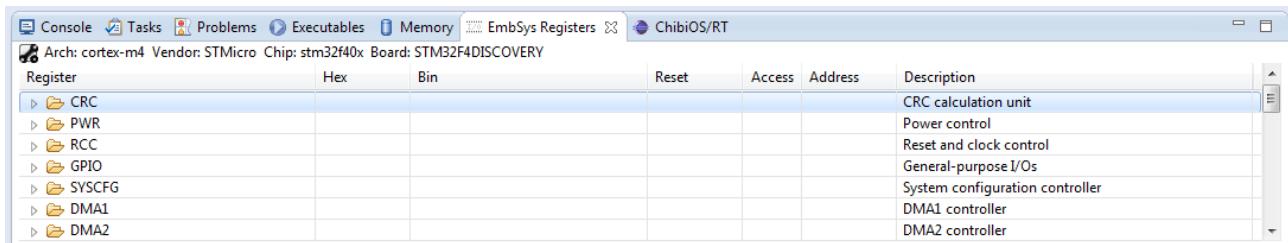
En aquest cas seleccioneu **Window → Preferences** i a dins seleccioneu:

**C/C++ → Debug → EmbSys Register View**

Ompliu les quatre caselles amb les opcions que s'indiquen:

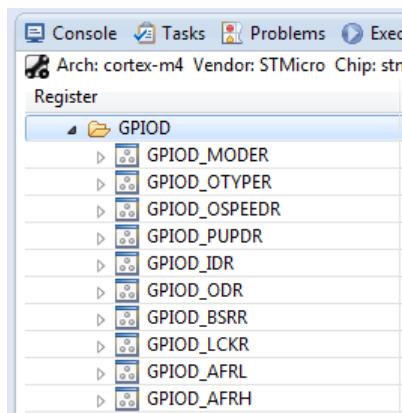


Si la configuració es correcta, la pestanya EmbSys Register mostrarà un aspecte com el següent:

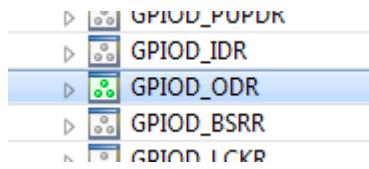


Podeu veure tots els registres interns del microcontrolador agrupats per les seves funcionalitats.

Per exemple, si obriu la carpeta GPIOD podreu veure tots el registres associats a aquest perifèric:



El registre ODR, per exemple, permet canviar la informació de sortida del port GPIOD. En seleccionar-lo (el seu color canvia a verd) podem veure el seu contingut.



A mode de exemple es mostra el contingut de GPIOD\_ODR

> GPIOD_IDR				0x00000000	R	0x40020C10	GPIO port input data register
> GPIOD_ODR	0x00001FCF	00000000000000000000000000111111001111	0x00000000	RW	0x40020C14	GPIO port output data register	
> GPIOD_BSRR	- write only -	----- write only -----	0x00000000	W	0x40020C18	GPIO port bit set/reset register	

En el cas de ODR, es pot desplegar el seu contingut:

Register	Hex	Bin	Reset	Access	Address	Description
GPIOD_ODR	0x00001FCF	00000000000000000000000000111111001111	0x00000000	RW	0x40020C14	GPIO port output data register
IO Pins (bits 0-15)	0x1FCF	0001111111001111				All GPIO Port D output pins in one package.
Pin_0 (bit 0)	0x1	1				
Pin_1 (bit 1)	0x1	1				
Pin_2 (bit 2)	0x1	1				
Pin_3 (bit 3)	0x1	1				
Pin_4 (bit 4)	0x0	0				CS43L32 Reset On
Pin_5 (bit 5)	0x0	0				
Pin_6 (bit 6)	0x1	1				
Pin_7 (bit 7)	0x1	1				
Pin_8 (bit 8)	0x1	1				
Pin_9 (bit 9)	0x1	1				
Pin_10 (bit 10)	0x1	1				
Pin_11 (bit 11)	0x1	1				
Pin_12 (bit 12)	0x1	1				Green Led On
Pin_13 (bit 13)	0x0	0				Orange Led Off
Pin_14 (bit 14)	0x0	0				Red Led Off
Pin_15 (bit 15)	0x0	0				Blue Led Off
GPIOD_BSRR	- write only -	----- write only -----	0x00000000	W	0x40020C18	GPIO port bit set/reset register
GPIOD_LCKR			0x00000000	RW	0x40020C1C	GPIO port configuration lock register
GPIOD_AFRL			0x00000000	RW	0x40020C20	GPIO alternate function low register

Fixeu-vos que fins i tot el plug-in té informació de quins perifèrics es troben connectats als ports GPIO dins la placa discovery com es el cas dels quatre leds.

El plug-in no només pot servir per veure el contingut dels registres sinó que també pot modificar-los. Si seleccionem, per exemple, el Pin 12 que correspon al led verd, podem canviar el seu estat.

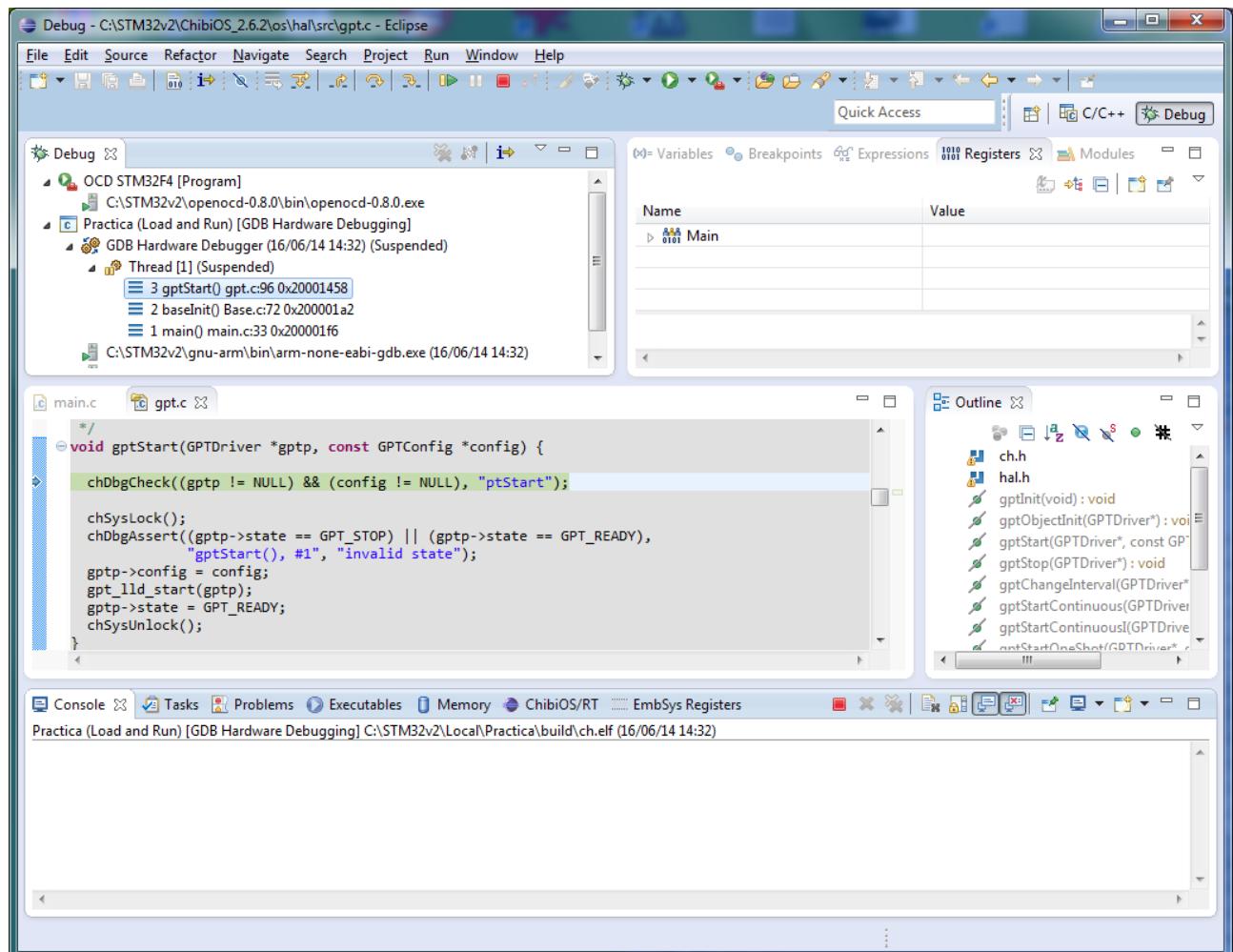
Pin_11 (bit 11)	0x1	1				Green Led On
Pin_12 (bit 12)	0x1	1				Orange Led Off
Pin_13 (bit 13)	0x0	0				Red Led Off
Pin_14 (bit 14)	0x0	0				Blue Led Off
Pin_15 (bit 15)	0x0	0				
GPIOD_BSRR	- write only -	-----	0x00000000	W	0x40020C18	GPIO port bit set/reset register

- Proveu de canviar l'estat d'alguns dels leds per comprovar la funcionalitat del plug-in. Verifiqueu que els leds de la placa fan els canvis corresponents.

## P1.10 Stack Frames

Quan esteu depurat un programa podeu observar que a la finestra de debug s'indiquen quines son les funcions que actualment no han acabat. A l'exemple següent, por després d'arrencar el programa es veu que hi ha corrents la funció **main()** que ha cridat a **baseInit()** que a la seva vegada ha cridat a la funció **gptStart()**.

Cada funció que ha començat però no ha acabat defineix un *Stack Frame*. Es a dir, té un conjunt d'elements ocupats a la pila que defineixen, per exemple, les seves variables locals.



La vista del Stack Frame pot ser interessant per veure quantes funcions estan en marxa al programa en un determinat instant.

Observeu que la finestra de Debug mostra que hi ha un fil d'execució "***Thread [1]***" que es troba associat al *stack frame*. Aquesta informació no es del tot correcta. Com ja hem dit fa poc, el nostre programa té dos fils d'execució: *idle* i *main* tal i com es veu pestanya **Chibios/RT** de la vista de Debug. Per tant, la vista del *stack frame* només ens donarà informació del fil que tenim actiu en el moment de parar el programa.

Observeu també que la vista del stack frame indica on es troba el codi de cada funció, el el cas de **main( )** podem veure a la figura que ens trobem a **0x200001F6**. Aquesta es la posició del comptador de programa abans de cridar-se la funció **baseInit( )**. Donat que la memòria RAM del microcontrolador comença a la direcció **0x20000000**, això ens demostra que estem executant a la RAM.

Amb això acaba la pràctica P1

## P2 - Accés al'LCD

La placa de pràctiques incorpora un LCD de dues files de 16 caràcters model MC1602C8-SYL fabricat per EVERBOUQUET. En aquesta pràctica aprendrem com accedir-hi.

### Objectius de la pràctica:

- Aprendre a programar la funcionalitat dels ports en mode sortida.
- Aprendre a fer servir un LCD basat en HD44780.

### Estudis Previs:

- EP 1 Verificació de marges de soroll a l'apartat P2.2
- Genereu el codi d'una primera versió de les funcions que es demanen a partir de P2.4

lcdGPIOInit	LCD_Backlight
lcdNibble	LCD_SendChar
LCD_SendString	LCD_ClearDisplay
LCD_Config	LCD_GotoXY

### Resultats:

- Funcions completes que es demanen pel fitxer **lcd.c**
- Ser capaç d'escriure els noms del membres del grup al LCD

### Resultats opcionals:

- Crear la funció **GPIO\_ModePushPull** que es demana a P2.4
- Ser capaç de generar caràcters especials com es demana a P2.7

### P2.1 Funcionament del LCD

El display LCD que fem servir, com la major part dels displays LCD en mode caràcter que hi ha al mercat, està basat en l'integrat Hitachi HD44780. Aquests és un circuit integrat que permet accedir des d'un processador a un LCD de fins a un màxim de 80 caràcters distribuïts en 1, 2 o 4 fileres.

La informació completa sobre el funcionament del'LCD la podeu trobar als fitxers "[LCD MC1602C8.pdf](#)" i "[HD44780U Driver.pdf](#)", en aquest apartat, però, es faran les explicacions mínimes necessàries per poder treballar-hi.

### Memòria interna

L'integrat HD44780 disposa internament de 144 Bytes de memòria RAM distribuïda en dos blocs: Un bloc *Display Data RAM* (DDRAM) de 80 bytes i un bloc *Character Generator RAM* (CGRAM) de 64 Bytes. Addicionalment, el xip disposa també de 9920 bits de memòria ROM denominada *Character Generator ROM* (CGROM).

NOM	TIPUS	MIDA	DESCRIPCCIÓ
DDRAM	RAM	80 Bytes	Contingut ASCII de la pantalla
CGRAM		64 Bytes	Caràcters personalitzats
CGROM	ROM	9920 bits	Caràcters estàndard

El bloc de memòria *Display Data RAM* (DDRAM) conté la informació del contingut de cada casella del display. El nostre display té dues línies de 16 caràcters. Per tant, té 32 posicions. Això vol dir que fem servir 32 dels 80 bytes de memòria DDRAM.

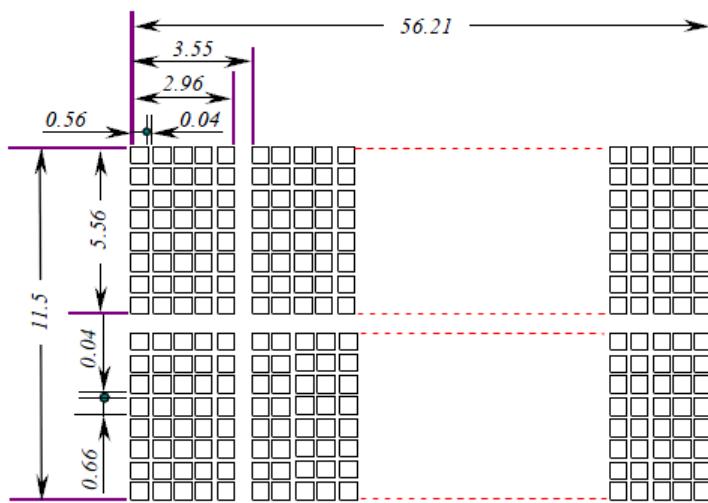
El bus de dades de la memòria DDRAM es de 7 bits (Direccions 0 a  $2^7 - 1 = 127$ ) per poder encabir els 80 bytes que conté, però l'organització interna de la memòria DDRAM es discontínua. La següent taula mostra els diferents rangs de direccions del bus que accedeix a la DDRAM i el seu contingut associat.

DIRECCIONS	Contingut
0 (0x00)	40 bytes Contingut de la primera filera i la tercera si n'hi ha
39 (0x27)	
40 (0x28)	No implementat
63 (0x3F)	
64 (0x40)	40 bytes Contingut de la segona filera i la quarta si n'hi ha
103 (0x67)	
104-127 (0x68-0x7F)	No implementat
127 (0x7F)	

El 80 bytes de DDRAM estan agrupat en dues àrees de 40 bytes. La primera, entre les direccions 0 i 39 contenen la informació de la primera filera (i la tercera si el display es de 4 línies). La segona àrea, entre les direccions 64 i 103 contenen la informació de la segona filera (i la tercera si el display es de 4 línies). Les direccions en els rangs 40-63 i 104-127 no estan implementades i, per tant, no es poden llegir ni escriure.

El nostre display té dues fileres de 16 columnes, per tant, la primera filera ocupa les direccions 0 a 15 i la segona filera ocupa les direccions 64 a 79. A cada posició, la memòria DDRAM emmagatzema el codi ASCII del caràcter a presentar.

El display està basat en una matriu de punts organitzada en caràcters com la representada a la figura següent:



Cada caràcter està format per una matriu de 8 fileres i 5 columnes (5x8). La última filera de cada caràcter es reserva normalment pel cursor i acostuma a estar en blanc. Alguns altres displays que fan servir el integrat HD44780 poden presentar també caràcters de 10 fileres i 5 columnes (5x10). La definició de la forma de cada caràcter està emmagatzemada en la memòria *Character Generator ROM* (CGROM) de 9920 bits. Els 9920 bits contenen la definició de 208 caràcters en format 5x8 i 32 caràcters en format 5x10. Les formes dels diferents caràcters continguts a la CGROM les podeu veure a partir de la [pàgina 18](#) del document "[HD44780U Driver.pdf](#)".

Els primers 8 codis ASCII de caràcter (0 a 7) són especials per què la seva definició no es troba a la CGROM sinó a la memòria *Character Generator RAM* (CGRAM). Això vol dir que podem definir fins a 8 caràcters especials amb la forma que nosaltres desitgem mentre sigui definida amb una matriu de 5x8. Cada caràcter d'usuari de 5x8 ocupa 8 posicions. Així, la definició del caràcter de codi 0 ocupa les posicions 0 a 7, les del caràcter 1 de la 8 fins a la 15 i així successivament fins el caràcter de codi 7 que ocupa les posicions 56 a 63. La següent figura mostra la construcció dels caràcters dins de la CGRAM.

CGRAM Address	Character Patterns (CGRAM data)
5 4 3 2 1 0 High      Low	7 6 5 4 3 2 1 0 High      Low
0 0 0	0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1
0 0 1	0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1
1 1 1	0 0 0 0 0 1 1 0 0 1 0 1 1 1 0 1 1 1

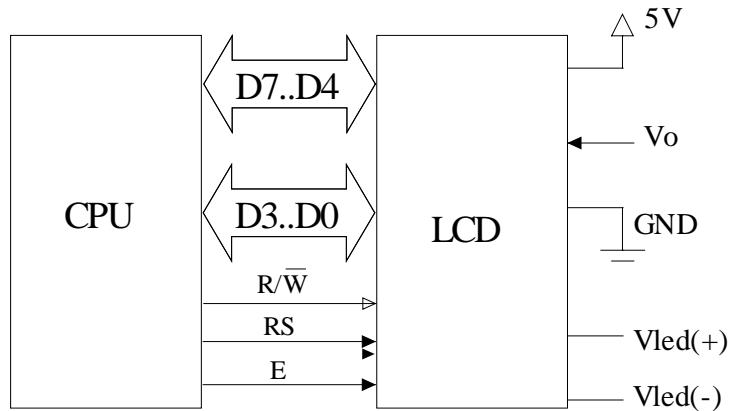
Observeu que els "1"s corresponen a les parts amb color diferent del fons i que els bits 7 a 5 de cada byte de la CGRAM no es fan servir mai.

Quan està en funcionament, l'LCD escombra de manera cíclica la seva matriu de punts controlant la seva encesa o apagada amb la informació del caràcter a presentar (DDRAM) i la forma d'aquest caràcter (CGROM i CGRAM).

### Interfície amb la CPU

Una vegada explicat el mecanisme que fa servir el driver per generar la imatge del LCD a partir de les seves memòries DDRAM, CGROM i CGRAM, explicarem el mecanisme de comunicació amb la nostra CPU.

El mòdul LCD es connecta a la resta del sistema amb un total de 16 línies, de les quals, 11 corresponen a la interfície de connexió amb el processador. La següent figura mostra les 16 connexions del mòdul LCD.

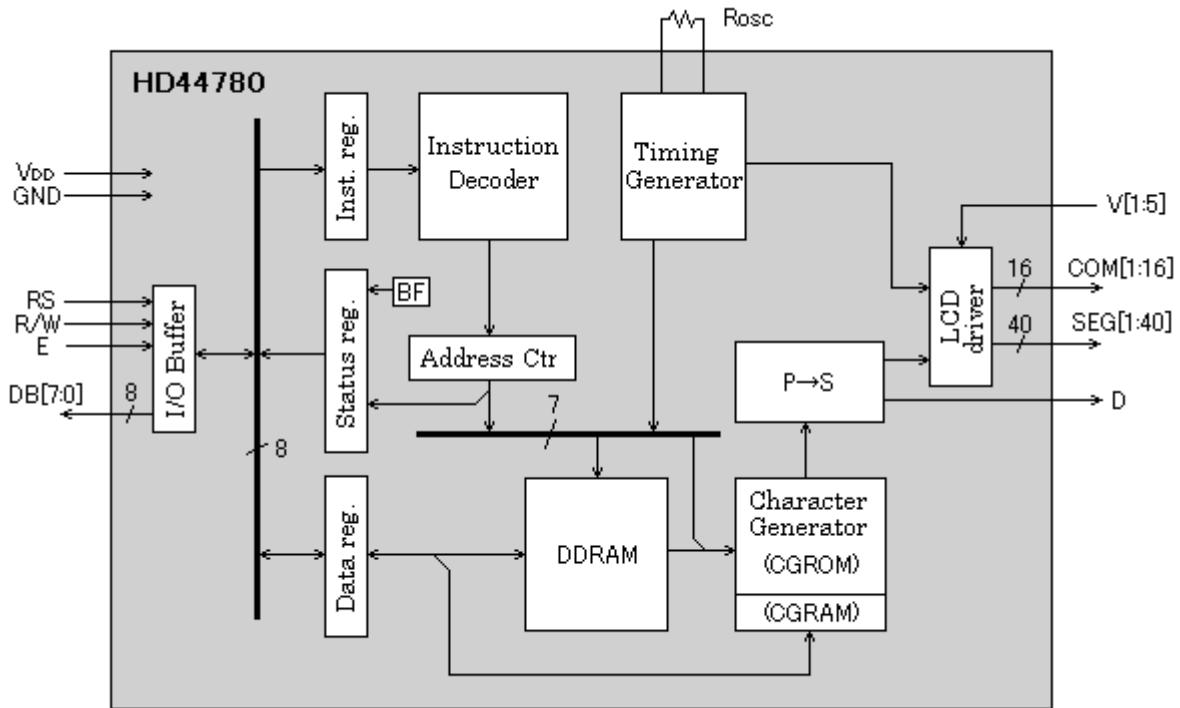


La descripció de les diferents línies es mostra a la taula següent:

TIPUS	NOM	DESCRIPCIÓ
General	5V	Alimentació positiva de 5 Volts
	GND	Referència de l'alimentació
	Vo	Control del contrast
	Vled(+)	Alimentació positiva del <i>backlight</i>
	Vled(-)	Alimentació negativa del <i>backlight</i>
Interfície	D7..D4	4 bits de la part alta del bus de dades
	D3..D0	4 bits de la part baixa del bus de dades
	R/W	Senyal de lectura/escriptura ("1" Lectura / "0" Escriptura)
	RS	Senyal <i>Register Select</i>
	E	<i>Strobe</i> de lectura i escriptura

La connexió més completa possible entre l'LCD i el processador fa servir 8 línies de dades D7...D0 i 3 línies de control R/W, RS i E. Aquesta interfície va ser dissenyada, en el seu moment, per connectar als busos externs de microprocessadors com els 8085, 6800 i Z-80 que operaven a baixes velocitats de rellotge (en el entorn de 1MHz). En el nostre cas farem servir els ports GPIO del microcontrolador STM32F407 de la nostra placa per accedir al'LCD ja que no disposem de bus extern habilitat i, si en tinguéssim, seria massa ràpid per l'LCD.

La següent figura mostra el diagrama de blocs intern simplificat del'integrat HitachiHD44780.



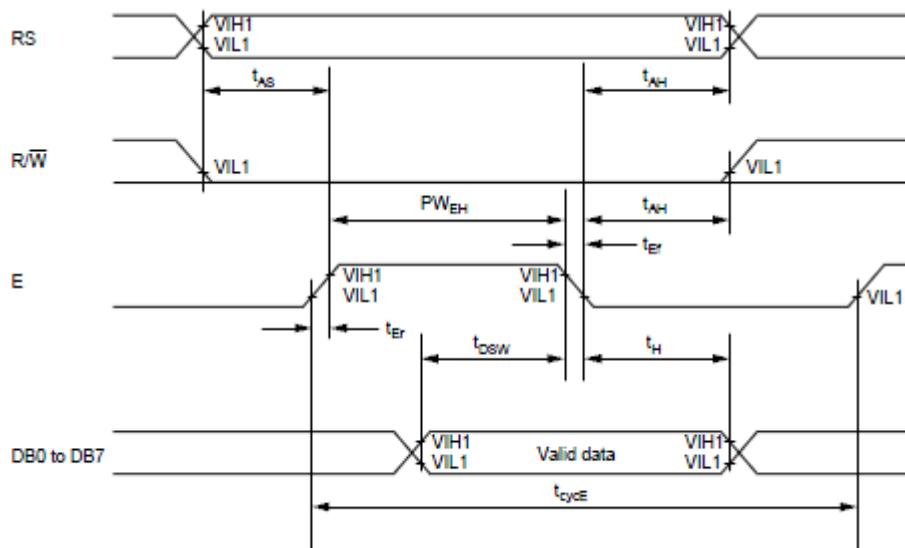
Les 11 línies de l'interfície DB7..DB0, RS, R/W i E es connecten a un *buffer* i aquest, amb un bus intern de dades de 8 bits. Aparentment,l'interfície no té bus de direccions, només bus de dades. En realitat, si que en té de bus de direccions, però només té una línia que es RS.

El senyal RS (*Register Select*) indica on ha d'anar la informació que es mou al bus de dades D7...D0. Si RS té un valor alt "1" les dades es llegeixen o escriuen al *Data Register* (DR). Si RS té un valor baix "0" les dades s'escriuen al *Instruction Register* (IR), però es llegeixen del *Status Register* (SR). Així el xip Hitachi HD44780U es comporta com si tingués dos registres en funció del valor de RS tal i com es mostra a la següent taula:

RS	R/W	Funcionalitat
0	0	Escrivim al <i>Instruction Register</i> (IR)
	1	Llegim l' <i>Status Register</i> (SR)
1	0	Escrivim al <i>Data Register</i> (DR)
	1	Llegim el <i>Data Register</i> (DR)

Observeu que l'*Instruction Register* (IR) no es pot llegir i que l'*Status Register* (SR) no es pot escriure.

Els cronogrames de lectura i escriptura del driver HD44780U, considerant RS com un únic bit de direccions, no són molt diferents dels cronogrames d'accés a una memòria. A mode d'exemple es mostra a continuació el cronograma per un cicle d'escriptura.



### Write Operation

Item	Symbol	Min	Typ	Max	Unit
Enable cycle time	$t_{cycE}$	500	—	—	ns
Enable pulse width (high level)	$PW_{EH}$	230	—	—	
Enable rise/fall time	$t_{er}, t_{fr}$	—	—	20	
Address set-up time (RS, R/W to E)	$t_{AS}$	40	—	—	
Address hold time	$t_{AH}$	10	—	—	
Data set-up time	$t_{DSW}$	80	—	—	
Data hold time	$t_H$	10	—	—	

Observeu que el senyal E (Enable) fa de senyal de validació (*strobe*) del cicle d'escriptura. La CPU posa el valor "0" o "1" a RS depenent de si volem accedir al registre d'instruccions o al registre de dades respectivament i indica, posant a "0" el senyal R/W que es un cicle d'escriptura. A continuació, amb un temps mínim de set-up  $t_{AS}$  s'activa el senyal Enable. Seguidament es posen les dades DB0..DB7 i, amb un temps de set-up mínim  $t_{DSW}$  es desactiva Enable per validar l'escriptura.

Com hem observat, el driver HD44780U es veu, des del punt de vista de la CPU, com dos registres que es poden llegir o escriure. Com podem llavors accedir als 144 bytes de memòria RAM que inclou el driver?

Les memòries internes del driver DDRAM i CGRAM estan connectades a un bus intern de direccions de 7 bits. Aquest bus es genera amb un comptador intern *Address Counter*. El registre *Data Register* (DR) està en tot moment connectat a la memòria DDRAM o la memòria CGRAM. Quan escrivim sobre DR, les dades passen a la posició de la DDRAM o CGRAM apuntada pel *Address Counter*. El comptador *Address Counter* s'incrementa automàticament cada vegada que s'escriu una nova dada al *Data Register* (DR).

Així, doncs, si *Address Counter* està apuntant actualment a la posició 64 de la DDRAM (inici de la segona filera) i escrivim dues dades 0x41, 0x42 consecutives al *Data Register* (DR), el valor 0x41 (codi ASCII de la lletra 'A') anirà a la direcció 64 i el valor 0x42 (codi ASCII de la lletra 'B') anirà a la direcció 65 i podrem llegir "AB" al principi de la segona filera del'LCD.

De la mateixa manera, quan llegim el *Data Register* (DR), en realitat estem llegint la posició de la memòria RAM apuntada per l'*Address Counter*. El comptador també s'incrementa automàticament en les operacions de lectura com ho fa a les operacions d'escriptura.

El comptador de 7 bits *Address Counter* es pot llegir en tot moment mitjançant els bits 6 a 0 del registre *Status Register*, accessible en lectura quan RS="0". El comptador, no obstant, no es pot escriure directament, per canviar el seu contingut, s'ha de fer servir ordres especials mitjançant l'*Instruction Register* (IR).

El driver del'LCD no pot rebre nova informació mentre està processant dades de l'última ordre rebuda. El senyal *Busy flag* (BF) accessible com a bit 7 del *Status Register* (SR) permet indicar si el driver està preparat per rebre noves ordres.

El registre *Instruction Register* (IR) ens permet donar ordres al driver del'LCD. Com que és un registre de 8 bits, hi ha un total de 256 ordres possibles. La major part de les possibles ordres estan relacionades amb el registre intern *Address Counter* de 7 bits. La següent taula mostra totes les ordres que es poden enviar al *Instruction Register* (IR). Com que aquest registre només és accessible en escriptura quan RS=0, a totes les fileres s'indica RS=R/W="0"

Instruction	Code										Description	Execution Time (max) (when $f_{\text{sp}}$ or $f_{\text{osc}}$ is 270 kHz)	
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0			
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.		
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms	
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 $\mu$ s	
Display on/off control	0	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 $\mu$ s
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 $\mu$ s	
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 $\mu$ s	
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 $\mu$ s	
Set DDRAM address	0	0	1	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 $\mu$ s							

Per cada ordre, a la taula s'indica també el temps màxim que cal per executar-la.

Les ordres es reconeixen dependent de la posició del primer "1" en el bus de dades DB7..DB0.

Així, l'ordre "*Set DDRAM address*" requereix DB7="1", l'ordre "*Set CGRAM address*" requereix DB7,6="01", i així fins l'ordre "*Clear Display*" que correspon a DB7...0="00000001".

Les dues últimes ordres permeten fixar el valor del *Address Counter*.

L'ordre "*Set DDRAM address*" fixa el valor del *Address Counter* per accedir a la memòria DDRAM. Com ja sabem, ens calen 7 bits per fixar una posició dins de la memòria DDRAM. Aquests corresponen als bits DB6...DB0 de l'ordre. Qualsevol posterior accés al registre de dades *Data Register* (DR) permetrà accedir a la DDRAM a partir de la posició indicada a l'ordre.

L'ordre "*Set CGRAM address*" fixa el valor del *Address Counter* per accedir a la memòria CGRAM. Aquesta es una memòria de 64 bytes, per tant, ens calen 6 bits. Aquests corresponen als bits DB5...DB0 de l'ordre. Qualsevol posterior accés al registre de dades *Data Register* (DR) permetrà accedir a la CGRAM a partir de la posició indicada a l'ordre.

A part de les dues ordres anteriors, hi ha dues ordres més a la llista que és important descriure:

L'ordre "*Display on/off control*" permet configurar 3 flags importants:

El bit DB2 (D) activa "1" o desactiva "0" tot el display.

El bit DB1 (C) activa "1" o desactiva "0" el cursor a la posició actual.

El bit DB0 (B) activa "1" o desactiva "0" d'intermitència del caràcter a la posició del cursor.

L'ordre "*Clear Display*" omple la memòria de caràcters 0x20 (Espai) i posa el comptador de direccions al principi de la memòria DDRAM.

La resta d'ordres té una certa importància en les operacions d'inicialització del display, però no són tan importants com les quatre anteriors.

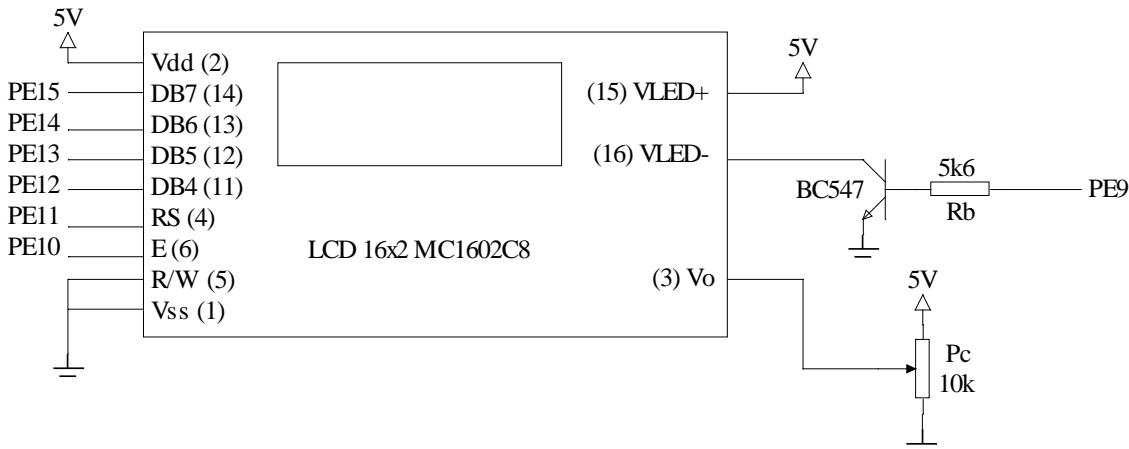
## **P2.2 Implementació a la placa de pràctiques**

La connexió completa entre el driver HD44780 i la CPU requereix, com s'ha indicat, 11 línies dividides en 8 línies de dades i 3 línies de control. El disseny del driver permet, però, fer servir només 6 línies per minimitzar el nombre de connexions necessàries entre el driver i la CPU.

Les condicions necessàries per treballar només amb 6 línies són:

- No fer servir operacions de lectura (R/W sempre "0"). Això vol dir que no podrem accedir al registre *Status Register*, ni podrem llegir la DDRAM o CGRAM.
- No fer servir les línies de dades DB3...DB0. Per enviar un Byte sencer (8 bits) farem dues transferències amb les línies DB7..DB4.

La següent figura mostra les connexions entre l'LCD i la CPU a la placa de pràctiques extreta del document "[Esquemes.pdf](#)".



Observeu que fem servir una interfície amb 6 línies DB7...DB4, RS i E.

A part de la interfície de comunicació, observeu que la línia **Vo** que controla el contrast de la pantalla està connectada al potenciómetre **Pc** (Amb una marca de color verd a la placa). Observeu també que la llum led de *backlight* està connectada al pin PE9 mitjançant un transistor NPN. D'aquesta manera podrem controlar la llum del LCD des de la CPU.

Totes aquests connexions entre el LCD i els PADs GPIO del microcontrolador es troben descrits al *board file .h* com *labBoard12.h*.

L'LCD que fem servir està alimentat a 5V i el microcontrolador STM32F407, en canvi, està alimentat a 3V. Donat que mai no llegim l'LCD (*R/W*="0") mai no arribarà 5V a les entrades del STM32F407 (tot i que ho podrien suportar). En tot cas hem de comprovar que els nivells lògics de la CPU i el LCD siguin compatibles en les operacions d'escriptura al LCD.

**EP 1:** Comproveu si els nivells lògics estàtics son compatibles entre el microcontrolador i l'LCD.  
En cas de que ho siguin, determineu els marges de soroll.  
La informació necessària es als documents "LCD MC1602C8.pdf" i "STM32F407 Datasheet.pdf" (pag. 100). Els ports que fem servir són de tipus CMOS.

## Inici del treball de laboratori

En aquest punt s'inicia el treball de laboratori. Això no vol dir que no hi hagi més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

⚡ Abans que res, comproveu el bon funcionament de la placa.

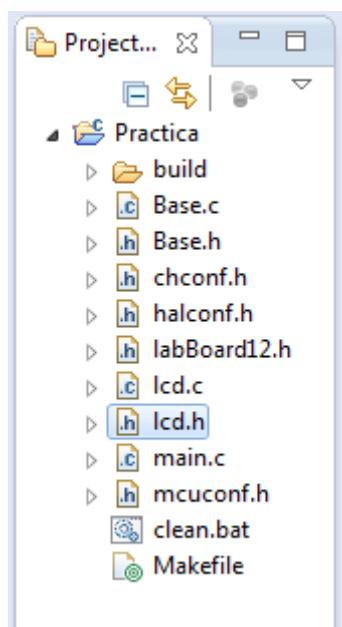
### P2.3 Codi modular i tipus de dades

La gestió del LCD la podríem incloure a dins del fitxer **main.c** de la pràctica anterior, però no seria una manera endreçada de treballar en un projecte en C. Tot el codi que desenvolupem a les pràctiques el podem incloure al mateix projecte **practica** que vam crear a la pràctica P1 però si no volem acabar amb un fitxer **main.c** il·legible, convé separar el codi en mòduls funcionals.

Cada mòdul funcional en C consta de dos fitxers, un fitxer **.c** que porta el codi i un fitxer de capçalera **.h**. Com que el nostre mòdul gestiona l'LCD, anomenarem aquests fitxers **lcd.c** i **lcd.h**.

Per fer la pràctica P2, la definició bàsica d'aquests fitxers ja us la donem a dintre del fitxer **Practica-P2.zip**.

- ▀ Assegureu-vos que esteu fora d'Eclipse, obriu el fitxer **Practica-P2.zip** i poseu els dos fitxers **lcd.c** i **lcd.h** a dins del directori **practica**, juntament amb main.c i els altres fitxer de la pràctica P1.
- ▀ Entreu ara a Eclipse i feu servir la perspectiva C/C++. Mireu a la subfinestra del explorador del projecte si podeu veure els nos fitxers **lcd.c** i **lcd.h**. En cas de que no es vegin, seleccioneu **Refresh** en el menú contextual del explorador de projecte o pitgeu **F5**. El contingut del projecte hauria de quedar com es mostra a la figura:



Ara ja podeu accedir i modificar els fitxers **lcd.c** i **lcd.h**, però encara no estan inclosos dins del nostre projecte. Per que hi estiguin, hem de modificar el fitxer *Makefile*.

- Obriu el fitxer *Makefile* i busqueu la llista de fitxers .C del projecte i afegiu el fitxer **lcd.c** a sota de **Base.c** com es mostra a la figura següent. A continuació salveu el fitxer *Makefile* modificat.

```
# The source .c files that make the project can be added
# between Base.c and main.c
# The character "\" at the end of all but the last line is important
PCSRC = Base.c \
    lcd.c \
    main.c
```

Amb aquest canvi, fem que el compilador compili també el contingut de **lcd.c** generant un fitxer objecte binari compilat **lcd.o**. Aquest fitxer s'ajuntarà fent servir el *linker* amb la resta de fitxers .O del projecte (un per cada fitxer en C o Assembler) per generar el fitxer binari **ch.elf** final del nostre projecte. Tot i que no ho sembla, el nostre projecte té ara 58 fitxers objecte, que esdevindran 59 en afegir **lcd.o** la propera vegada que compilem. D'aquests 58 fitxers, dos corresponen als fitxers **main.c** i **Base.c** del nostre projecte i la resta son del sistema operatiu ChibiOS/RT que, en aquest moment, només fem servir per inicialitzar la placa.

La llista de tots aquests fitxers objecte la podeu trobar dins de la carpeta **build → obj** del explorador de projectes.

- Per completar la inclusió del mòdul al projecte, hem de incloure una crida al fitxer de capçalera **lcd.h** dins de cada fitxer .c que en faci crides a les seves funcions. Com que nosaltres volem fer servir el LCD dintre de **main.c**, afegirem **#include "lcd.h"** al principi del fitxer **main.c** just a sota de la inclusió de **Base.h**.

```
main.c
=====
main.c

Practica 1
=====

#include "Base.h"      // Basic definitions
#include "lcd.h"        // LCD module header file
```

- Si tornem a compilar amb Ctrl+B, trobarem que ara es compila el fitxer **lcd.c**

```

Problems Tasks Console Properties
CDT Build Console [Practica]
Compiling ../ChibiOS_2.4.0/os/hal/platforms/STM32/GPIOv2/pal_lld.c
Compiling ../ChibiOS_2.4.0/boards/ST_STM32F4_DISCOVERY/board.c
Compiling ../ChibiOS_2.4.0/os/various/lis302dl.c
Compiling lcd.c
Compiling Base.c
Compiling main.c
Linking build/ch.elf
Creating build/ch.hex
Creating build/ch.bin
Creating build/ch.dmp
Done

**** Build Finished ****

```

De moment però, **lcd.c** no te gaires continguts i no fa res al nostre projecte. El nostre treball pel que resta d'aquesta pràctica es omplir-lo de contingut. També podem omplir de contingut **lcd.h** si cal.

En general **lcd.c** contindrà:

- Definicions de variables associades al LCD (si n'hi ha)
- Codi de totes les funcions

Mentre que **lcd.h** contindrà:

- Definicions de constants i pseudofuncions mitjançant instruccions **#define**
- Prototips de les funcions públiques

Podeu mirar el contingut de **Base.h** i **Base.c** per veure un exemple d'aquesta separació de funcions.

## Tipus de dades

Abans de començar a omplir el contingut de **lcd.c** convé explicar els tipus de dades que podem fer servir en els nostres programes. De moment no farem servir variables en coma flotant, tot i que la CPU que en fem servir té unitat aritmètica de coma flotant, per que no ens calen. Només farem servir, per tant, tipus sencers. En llenguatge C s'acostuma a fer servir variables de tipus **short int**, **int** i **long int** per diferents nombres de bits dels tipus sencers. Desafortunadament, en cada CPU aquestes definicions poden voler dir coses diferents. Per evitar problemes nosaltres farem les següents definicions de tipus que no tenen ambigüïtats:

TIPUS	MIDA	Signe
uint8_t	8 bits	NO
int8_t, char	8 bits	SI
uint16_t	16 bits	NO
int16_t	16 bits	SI
uint32_t	32 bits	NO
int32_t	32 bits	SI

El tipus **int8\_t** que correspon a un Byte amb signe, per motius de compatibilitat, quan el fem servir com a definició de caràcter, li direm **char**.

No hem d'oblidar que la nostra CPU es de 32 bits, això vol dir que el tipus de dades que pot fer servir de manera més eficient es **int32\_t** o **uint32\_t**. Per tant, si no tenim un problema de falta de RAM per que hem ocupat els 192 kBytes que en té el processador, serà el nostre tipus preferit.

## **P2.4 Configuració dels ports**

La primera funcionalitat que hem d'aconseguir es programar les 7 línies associades al'LCD (DB7..DB4,RS,E,BL) com a ports de sortida. A la pràctica P1 ja vam fer servir els 4 leds, però no ens va caldre programar-los com a sortida per que el codi d'inicialització de la placa ja ho fa. L'LCD es extern a la placa STM32F4 Discovery i, per tant, no s'inicialitzen els seus ports. Gairebé tots els microcontroladors deixen els ports en estat d'entrada després del **reset** ja que es la opció més segura, i el nostre no n'és una excepció.

El microcontrolador, per tant, arrenca amb totes les línies GPIO en mode entrada. Quan cridem *baseInit( )* al principi de *main( )*, el primer que fa aquesta funció es cridar *halInit( )* que fa la inicialització del hardware i, en concret, dels ports de la placa. La configuración que es fa de tots els ports es troba als fitxers *board.c* i *board.h* que podeu trobar al directori:

C:\STM32v2\ChibiOS\_2.6.2\boards\ST\_STM32F4\_DISCOVERY

Examinant el fitxer *board.h* podem veure que els pins associats als LEDS (PD12 a PD15) es configuren en mode sortida Push-Pull. També podem veure que tots els pins que no es fan servir a la placa discovery en els ports A fins D, es configuren en mode entrada amb pull-up i amb els bits corresponents del registre ODR a "1". En el cas dels pins lliures del port E, entre els que s'inclouen els pins del LCD, es configuren en mode entrada sense pull-up però també am els bits del registre ODR a "1".

Per tant, hem de canviar la configuració dels pins del LCD des del estat d'entrada amb pull-up després de la crida a *baseInit( )* al estat de sortida *Push-Pull*.

Com ja es va indicar a la pràctica P1, cada port de GPIO està associat a 10 registres de 32 bits que son accessibles mitjançant una estructura **GPIO\_TypeDef**. Per poder configurar els ports, ens cal treballar, al menys, amb 4 d'aquests registres: MODER, OTYPER, OSPEEDR y PUPDR.

A la pràctica 1 no ens va caldre programar aquests registres degut a que el procés d'arrencada del sistema que es fa amb dins de *baseInit( )* ja s'encarrega de inicialitzar tots els perifèrics de la placa STM32F4Discovery. L'LCD, però, és extern a aquesta placa, i, per tant, hem d'inicialitzar manualment la configuració dels ports que fa servir.

La definició completa de la programació dels ports GPIO es troba a partir de la **pàgina 136** del document "[STM32F4 Reference.pdf](#)". A continuació fem un resum de les funcionalitats de programació.

## Registre MODER

Aquest registre es pot accedir com a variable **GPIOx->MODER** on x es la lletra del port A..E,H.

### 6.4.1 GPIO port mode register (GPIOx\_MODER) (x = A..I)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]	MODER14[1:0]	MODER13[1:0]	MODER12[1:0]	MODER11[1:0]	MODER10[1:0]	MODER9[1:0]	MODER8[1:0]								
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[1:0]	MODER6[1:0]	MODER5[1:0]	MODER4[1:0]	MODER3[1:0]	MODER2[1:0]	MODER1[1:0]	MODER0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

- 00: Input (reset state)
- 01: General purpose output mode
- 10: Alternate function mode
- 11: Analog mode

El registre de 32 bits té 2 bits associats a cada una de les 16 línies del port. Així, els bits 0,1 controlen la línia Px0, els bits 2,3 la línia Px1 i així successivament. Per tant, els bits que controlen la línia n (0...15) són els bits 2n i 2n+1. Les possibles funcionalitats depenen del valor de n són:

Bit 2n+1	Bit 2n	Funcionalitat
0	0	Entrada (Per defecte després de Reset)
0	1	Sortida
1	0	Funcionalitat alternativa
1	1	Mode analògic

Les dues primeres opcions són els modes normals de GPIO d'entrada o sortida.

La tercera opció dóna la possibilitat de connectar algun altre perifèric del microcontrolador a aquesta línia, en lloc de un GPIO genèric. Finalment, el mode analògic es per fer servir els ADCs i DACs que incorpora el microcontrolador.

En el nostre cas hem de posar les 7 línies en mode de sortida.

## Registre OTYPER

Aquest registre es pot accedir com a variable **GPIOx->OTYPER** on x es la lletra del port A..E,H.

### 6.4.2 GPIO port output type register (GPIOx\_OTYPER) (x = A..I)

Address offset: 0x04

Reset value: 0x0000 0000

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

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 OTy[1:0]: Port x configuration bits (y = 0..15)

These bits are written by software to configure the output type of the I/O port.

0: Output push-pull (reset state)

1: Output open-drain

D'aquest registre tenim un bit per cada línia de GPIO, els bits 16 a 31 no s'han de tocar i s'han de deixar a "0". Aquest registre només afecta la línia quan opera en mode de sortida, les funcions possibles són:

Bit n	Funcionalitat
0	Sortida Push-Pull (Per defecte després de Reset)
1	Sortida en drenador obert

Si posem "0" operem en mode Push-Pull, es a dir, forcem tant els "1"s com els "0"s. Si posem "1" operem en drenador obert, es a dir, forcen els "0", però deixem la línia en alta impedància quan hi ha un "1" a la sortida.

Nosaltres volem sortides en mode Push-Pull, per tant, de fet, com que és el estat per defecte després de Reset, i després de cridar *baseInit( )* no cal tocar el registre OTYPER.

## Registre OSPEEDR

Aquest registre es pot accedir com a variable **GPIOx->OSPEEDR** on x es la lletra del port A..E,H.

### 6.4.3 GPIO port output speed register (GPIOx\_OSPEEDR) (x = A..I)

Address offset: 0x08

Reset values:

- 0x0000 00C0 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15[1:0]	OSPEEDR14[1:0]	OSPEEDR13[1:0]	OSPEEDR12[1:0]	OSPEEDR11[1:0]	OSPEEDR10[1:0]	OSPEEDR9[1:0]	OSPEEDR8[1:0]								
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
OSPEEDR7[1:0]	OSPEEDR6[1:0]	OSPEEDR5[1:0]	OSPEEDR4[1:0]	OSPEEDR3[1:0]	OSPEEDR2[1:0]	OSPEEDR1[1:0]	OSPEEDR0[1:0]								
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 OSPEEDR<sub>y</sub>[1:0]: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O output speed.

00: 2 MHz Low speed

01: 25 MHz Medium speed

10: 50 MHz Fast speed

11: 100 MHz High speed on 30 pF (80 MHz Output max speed on 15 pF)

Igual que el registre MODER, tenim 2 bits para cada línia del port. Igualment els bits es troben a les posicions 2n i 2n+1 per cada línia del port.

Aquest registre controla la velocitat del port. Quan més alta sigui, més ràpids podran ser els senyals que hi passin, però també serà més alt el consum. Les possibles funcions són:

Bit 2n+1	Bit 2n	Funcionalitat
0	0	2 MHz (Per defecte després de Reset)
0	1	25 MHz
1	0	50 MHz
1	1	100 MHz

El microcontrolador arrenca amb aquest registre a zero (2 MHz) per garirebé tots els ports. El contingut de *board.h*, però, fa que els pins lliures de la placa discovery es configuri en el mode de màxima velocitat de 100MHz.

L'LCD és de baixa velocitat, per tant, amb 2MHz en tenim prou. Tot i això funcionarà correctament també a 100MHz, per tant, no cal tocar el registre OSPEEDR. Tot i que seleccionar 2MHz reduiria el soroll generat per les línies del LCD.

## Registre PUPDR

Aquest registre es pot accedir com a variable **GPIOx->PUPDR** on x es la lletra del port A..E,H.

### 6.4.4 GPIO port pull-up/pull-down register (GPIOx\_PUPDR) (x = A..I)

Address offset: 0x0C

Reset values:

- 0x6400 0000 for port A
- 0x0000 0100 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw										
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw										

Bits 2y:2y+1 **PUPDRy[1:0]: Port x configuration bits (y = 0..15)**

These bits are written by software to configure the I/O pull-up or pull-down

00: No pull-up, pull-down

01: Pull-up

10: Pull-down

11: Reserved

Igual que els registres MODER i OSPEEDR, tenim 2 bits para cada línia del port. Igualment els bits es troben a les posicions  $2n$  i  $2n+1$  per cada línia del port.

Aquest registre controla les resistències de *pull-up / pull-down* al pin de sortida del port.

Les possibles funcions son:

Bit 2n+1	Bit 2n	Funcionalitat
0	0	No Pull-up ni Pull-down (Per defecte després de Reset)
0	1	Pull-up
1	0	Pull-down
1	1	Combinació prohibida

Com que al'LCD no volem fer servir ni *Pull-Up* ni *Pull-Down*, podem deixar l'estat per defecte després de cridar a *baseInit( )* i no tocar aquest registre. Observeu que per totes les altres línies lliures a ports que no siguin "E", es configura *Pull-Up* per tots els pins lliures.

Al final veiem que, en realitat, no cal tocar garireé cap registre. Només es estríctament necessari canviar el valor del registre MODER dels pins associats al'LCD.

- ⚠ Vigileu que només heu de canviar els bits de les línies associades al'LCD, la resta de bits del registre les heu de deixar al mateix estat que estiguin, el qual, potser, no és el de *reset*. Això vol dir que CAL fer servir màscares AND i OR.

Recordeu, del contingut del fitxer *board.c*, que totes les línies que no es fan servir a la placa discovery es configuren amb el seu bit ODR a "1". L'estat per defecte de totes les línies del LCD hauria de ser "0", per tant, afegirem, dins de *lcdGPIOinit* el codi necessari per posar totes les sortides del LCD a "0".

#### **EP 2:** Elaboreu una versió preliminar de la funció **lcdGPIOInit**.

- Ompliu de contingut la funció **lcdGPIOInit** del fitxer **lcd.c** per tal que les 7 línies associades al LCD es configuri en mode de sortida **Push-Pull** amb nivell de sortida baix. Aquesta funció ja està definida al fitxer **lcd.c**, només heu d'omplir el seu contingut.
- ¶ Per omplir la funció **lcdGPIOInit** només cal modificar els registres MODER i ODR (directament o mitjançant el registre BSRR), no obstant, una manera més genèrica de treballar es crear una funció:

**GPIO\_ModePushPull(GPIO\_TypeDef \*port, int32\_t linea)**

que permeti programar com sortida *Push-Pull* amb sortida baixa qualsevol línia de qualsevol port. Després faríem crides a aquesta funció dins de **lcdGPIOInit**.

Per tal de que la funció funcioni a tots els ports i no depengui del estat previ dels ports, hauríem de fixar també els valors adequats dels registres OTYPER, OSPEEDR i PUPDR.

#### **P2.5 Gestió del *backlight***

La llum de *backlight* permet veure millor el display quan hi ha poca llum. Tal i com s'ha vist, aquesta llum s'activa mitjançant un transistor NPN que té la seva base a PE9. La gestió de PE9 com a port de sortida és idèntica a la que s'ha fet amb els leds a la pràctica P1.

Per poder verificar el bon funcionament de la funció de programació dels ports, podem implementar i provar la funció **LCD\_Backlight(int32\_t on)**.

Aquesta funció ha d'encendre el *backlight* si **on** és cert, i apagar-lo si és fals.

#### **EP 3:** Elaboreu una versió preliminar de la funció **LCD\_Backlight**.

- Ompliu de contingut la funció **LCD\_Backlight** del fitxer **lcd.c**

Per provar la funció podem modificar la funció **main( )** de manera que quedi:

```
int main(void)
{
    baseInit();      // Basic initialization
    lcdGPIOInit();   // Program the GPIO I/O lines
    LCD_Backlight(1); // Turn on LCD backlight
    SLEEP_MS(2000);   // Wait 2s
    LCD_Backlight(0); // Turn off LCD backlight
    while (1);        // Infinite loop so we don't exit main()
}
```

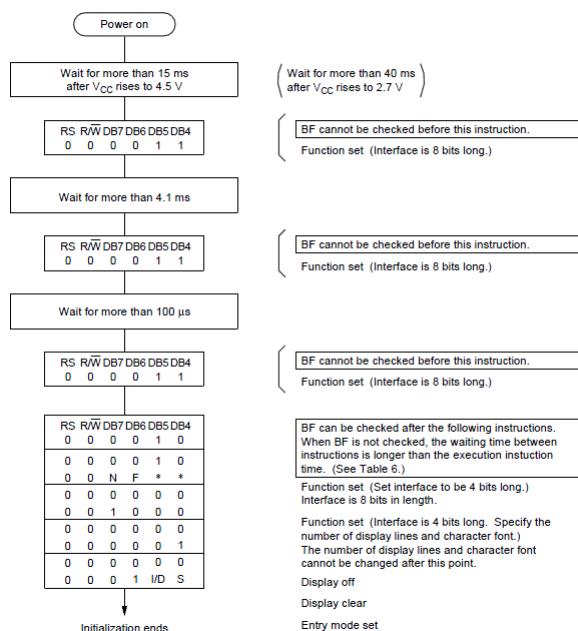
Si tot funciona correctament, el *backlight* s'hauria d'encendre per apagar-se després de 2 segons. Observeu que després de *lcdGPIOInit()* s'apaga el *Backlight*. Recordeu que tots els pins lliures de la placa *discovery* es configuren amb els bits de ODR a "1".

- Proveu el programa anterior per verificar les funcions **lcdGPIOInit** i **LCD\_Backlight**

## P2.6 Inicialització del display

El següent pas que hem de fer és inicialitzar el display. L'LCD, per defecte, arranca en mode de 8 bits fent servir DB7...DB0. Per poder treballar en mode de 4 bits hem de fer una seqüència d'inicialització que correspon a escriure unes dades concretes al LCD fent servir una temporització correcta.

El procediment d'inicialització del driver del LCD en mode de 4 bits fent servir DB7..DB4 es troba a la pàgina 47 del document "[HD44780U Driver.pdf](#)" i es mostra a la figura següent.



No és una seqüència complexa, però s'ha de fer bé. Per simplificar la pràctica, el codi d'inicialització ja us el donem a la funció **LCD\_Init** que es troba al final d'**lcd.c**. Aquesta funció realitza una sèrie d'escriptures al LCD fent servir una funció externa **lcdNibble**. Aquesta nova funció **lcdNibble** està descrita a dins de **lcd.c**, però no té contingut. S'ha d'implementar. La funció **LCD\_Init** també crida la funció **lcdGPIOInit** que ja hem provat.

La funció **lcdNibble** ha de fer el cicle d'escriptura descrit a P2.1. Bàsicament, fixem el valor que desitgem a D7...D4 i RS i, a continuació, es fa un pols al senyal d'enable (E). Tot i que no cal tant temps, farem un pols de 10 $\mu$ s. És a dir, fixarem el valor de D7..D4 i RS, esperarem 10 $\mu$ s, posarem E a "1", esperarem 10 $\mu$ s més, retornarem E a "0" i esperarem uns altres 10 $\mu$ s.

La definició de la funció **lcdNibble** és:

```
void lcdNibble(int32_t nibbleCmd,int32_t RS)
```

**nibbleCmd** pot valdre de 0 a 15 i conté la informació a posar a DB7...DB4. Així si **nibbleCmd** val 3, això correspon en binari a "0011" i, per tant DB7="0", DB6="0", DB5="1", DB4="1". La variable d'entrada **RS** indica el valor que s'ha de posar a la línia RS. Si RS és cert, RS="1", si RS és fals, RS="0".

**EP 4:** Elaboreu una versió preliminar de la funció **lcdNibble**.

■ Ompliu de contingut la funció **lcdNibble** del fitxer **lcd.c**

Una vegada implementada la funció **lcdNibble**, ja podem inicialitzar el LCD. Reescriu la funció **main( )** de manera que sigui:

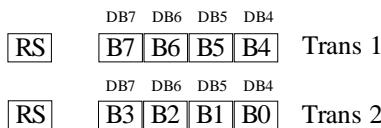
```
int main(void)
{
    baseInit();      // Basic initialization
    LCD_Init();     // Program the GPIO I/O lines
    while (1);      // Infinite loop so we don't exit main( )
}
```

■ Proveu el codi anterior, sabreu que ha funcionat correctament per que al display s'hi escriurà "OK"

## P2.7 Escrivint text al display i altres funcions

Per escriure text al display hem d'omplir de contingut dues funcions d'lcd.c, **LCD\_SendChar** i **LCD\_SendString**. **LCD\_SendChar** envia un caràcter al *Data Register* (DR), de manera que es copia a la DDRAM a la posició apuntada pel *Address Counter*. Recordeu que l'accés al *Data Register* es fa quan RS="1". Com que l'accés al LCD es fa mitjançant només 4 bits (DB7..DB4) i el nostre caràcter en té 8, B7...B0, haurem de fer dos transferències. A la primera farem la transferència dels 4 bits alts (nibble alt) del caràcter B7..D4 a DB7...DB4 mitjançant **lcdNibble**. A la segona, també mitjançant **lcdNibble**, farem la transferència del 4 bits baixos (nibble baix) del caràcter B3...B0 a DB7...DB4. En totes dues transferències farem servir un valor alt per RS.

Dada a transferir fent servir DB7...DB4



La funció **LCD\_SendString** té com argument una cadena (punter a caràcter) i fa tantes crides a **LCD\_SendChar** com caràcters tingui la cadena enviant un caràcter cada cop.

**EP 5:** Elaboreu una versió preliminar de les funcions **LCD\_SendChar** i **LCD\_SendString**.

■ Implementeu les funcions **LCD\_SendChar** i **LCD\_SendString**

■ Recordeu que l'LCD no pot rebre dades mentre està ocupat i que no podem saber si està ocupat amb la *Busy flag* per que no podem llegir-la. L'única manera de garantir que l'LCD està lliure quan ens intentem comunicar es garantir que el *driver* ha tingut temps de fer l'última operació. Totes les ordres del'LCD s'executen en un temps màxim d'uns 40µs, llevat de *Clear Display* i *Return Home* que ocupen una mica més de 1,5ms. No oblideu d'afegir el temps necessari al final de cada funció per evitar col·lisions.

Per provar les funcions només cal afegir una crida, com per exemple "**LCD\_SendString("Hello");**", després de "**LCD\_Init();**" dins de **main()**. Si ho fem, a la pantalla sortirà "OKHello\_

Les tres funcions que ens queden per omplir de contingut són:

**void LCD\_ClearDisplay(void);**

Esborra la pantalla i posa el cursor a 0,0

```
void LCD_Config(int32_t Disp,int32_t Cursor,int32_t Blink);
```

Configura el display

Si Disp és cert activa el display, si no, el desactiva

Si Cursorés cert mostra el cursor, si no, l'amaga

Si Blinkés cert activa la intermitència, si no, la desactiva

```
void LCD_GotoXY(int32_t col,int32_t row);
```

Posa el cursor a la posició indicada

col: Columna (0..LCD\_COLUMNS-1)

row: Filera (0..LCD\_ROWS-1)

**EP 6:** Elaboreu una versió preliminar de les funcionsanteriors.

■ Implementeu i ompliu de contingut les tres funcions que manquen per fer.

Amb això ja hem acabat el mínim de funcions que necessitem per treballar amb el LCD.

■ Feu una funció **main( )**que posi al display els vostres noms a les dues línies del display i ensenyeu-li al vostre professor de pràctiques.

### 💡 Generació de caràcters especials

Les funcionalitats del mòdul del LCD definides a **lcd.c** es poden ampliar molt. A mode d'exemple podeu generar una funció:

```
void LCD_CustomChar(int32_t pos,uint8_t *data)
```

que permeti generar caràcters especials pels codis ASCII 0 a 7. La variable pos indicaria el caràcter a definir (0..7) i el punter data apuntaria al primer del 8 Bytes necessaris per definir la forma del caràcter.

Per generar un caràcter s'ha d'apuntar a la memòria CGRAM fent servir l'ordre *Set CGRAM Address*. Després de definir un caràcter fora una bona idea tornar a apuntar a la memòria DDRAM fent servir *Set DDRAM Address*per evitar que una escriptura posterior no precedida d'**LCD\_ClearDisplay**o **LCD\_GotoXY**corrompi la memòria CGRAM.

Amb això acaba la pràctica P2

## P3 – Acceleròmetre 3D

La placa STM32F4 Discovery incorpora un acceleròmetre MEMS 3D capaç de proporcionar valors d'acceleracions en tres eixos ortogonals X, Y i Z. L'acceleròmetre de LIS302DL permet mesurar acceleracions de fins a +/- 8g i es comunica amb el microcontrolador amb un bus SPI.

### Objectius de la pràctica:

- Aprendre a treballar amb l'acceleròmetre LIS302DL.
- Entendre i fer servir el bus SPI.

### Estudis Previs:

- EP 1 – Configuració de rellotge SPI com es demana al apartat P3.2.
- EP2, EP3 – Escriure una versió preliminar de la funció initAccel segons es demana a P3.3 i P3.5.
- EP4 – Escriure una versió preliminar del programa final demanat a P3.6.

### Resultats:

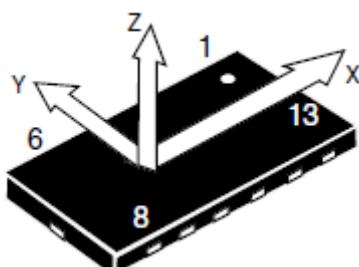
- Funció initAccel demandada.
- Programa complet que es demana a P3.6.

### Resultats opcionals:

- Crear la funció **writeAccel** que es demana a P3.5.
- Fer el procediment de reduir el soroll que es proposa a P3.6.

### P3.1 Funcionament de l'acceleròmetre i connexió amb el microcontrolador

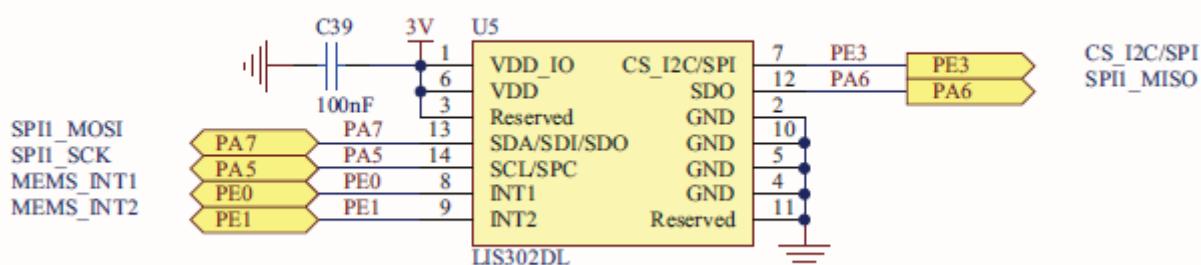
L'acceleròmetre es troba descrit al fitxer “[LIS302DL.pdf](#)”. Tal i com s'explica al apartat I3 de la introducció, es troba al centre de la placa STM32F4 Discovery entre els dos polsadors i rodejat pels leds. La següent figura mostra la distribució dels 3 eixos del'acceleròmetre.



Observant la posició del acceleròmetre a la placa Discovery podem veure que l'eix X va en la direcció baix-dalt, que uneix els dos connectors USB, l'eix Y va en la direcció del pulsador *Reset* al pulsador User i l'eix Z va en el sentit perpendicular a la placa.

## Connexions

Les connexions de la placa es troben al document “[Esquemes.pdf](#)”. A continuació es mostra el detall de les connexions del acceleròmetre.



Les connexions estan descrites a la següent taula

SENYAL	MCU PORT	DESCRIPCIÒ
SCL	PA5	SPI1 Clock : Relotge SPI generat pel MCU
MOSI	PA7	SPI1 MOSI : Master Out / Slave In
MISO	PA6	SPI1 MISO : Master In / Slave Out
CS	PE3	SPI Chip Select (Actiu baix)
INT1	PE0	Línia d'interrupció 1
INT2	PE1	Línia d'interrupció 2

Els quatre primers senyals configuren la comunicació entre el microcontrolador i el dispositiu mitjançant un bus SPI. Els últims dos senyals INT1 i INT2 es poden programar per que generin interrupcions quan algun eix rep un senyal d'acceleració prou fort tot i que no les farem servir en aquesta pràctica. Tota aquesta informació es troba al **board file** de la placa de pràctiques que es troba al projecte.

El pin PE3, com s'indica, es fa servir com a Chip Select, per tant, s'haurà de configurar com GPIO de sortida *Push-Pull* de manera que el seu valor de sortida el podrem escollir entre "1" (perifèric desactivat) i "0" perifèric activat.

Per poder fer servir els pins PA5, PA7, PA6 com a senyals SCL, MOSI i MISO del perifèric SPI1, s'ha de canviar el seu comportament per defecte com a GPIO a la funció alternativa de perifèric SPI1.

El apartat 3 del document "[STM32F407 Datasheet.pdf](#)" explica les possibles funcions que pot fer servir cada pin del microcontrolador per cada possible encapsulat. En el nostre cas fem servir l'encapsulat LQFP100 que es troba a la pàgina 39 d'aquest document. Com es pot veure a la taula 6, que es troba a partir de la pàgina 44 del document, cada pin del microcontrolador de tipus I/O, a part de poder fer-se servir com a GPIO, pot fer funcions alternatives.

A mode d'exemple, el pin PA5 es troba descrit a la pàgina 46 del document on s'indica que es troba associat al pin 30 de l'encapsulat LQPF100 i que a part de funcionar com a GPIO PA5 també pot actuar com rellotge del canal SPI1 (SPI1\_SCK) entre altres funcions.

**Table 6. STM32F40x pin and ball definitions (continued)**

LQFP64	LQFP100	LQFP144	LFBGA176	LQFP176	Pin number	Pin name (function after reset) <sup>(1)</sup>	Pin type	I/O structure	Notes	Alternate functions	Additional functions
					21 30 41 P4 51	PA5	I/O	TTa	(4)	SPI1_SCK/ OTG_HS_ULPI_CK / TIM2_CH1_ETR/ TIM8_CHIN / EVENTOUT	ADC12_IN5/DAC2_OUT

Cada pin de l'integrat pot tenir fins a 16 funcions diferents que es numeren des de AF0 a AF15 (AF vol dir *Alternate Function*). La llista de funcions alternatives associades a cada pin de GPIO es troba a la taula 7 del document que comença a la pàgina 56. Aquesta taula mostra que PA5, a part de funcionar com a GPIO, pot fer 5 funcions més, de les quals, la funció AF5 es SPI1\_SCK.

**Table 7. Alternate function mapping**

Port	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
	SYS	TIM1/2	TIM3/4/5	TIM8/9/10/11	I2C1/2/3	SPI1/SPI2/ I2S2/I2S2ext	SPI3/I2Sext/ I2S3	USART1/2/3/ I2S3ext	USART4/5/ USART6	CAN1/CAN2/ TIM12/13/14	ETH	FSMC/SDIO/ OTG_FS	DCMI			
PA5			TIM2_CH1 TIM2_ETR		TIM8_CHIN		SPI1_SCK					OTG_HS_ULPI_CK				EVENTOUT
PA6			TIM1_BKIN	TIM3_CH1	TIM8_BKIN		SPI1_MISO					TIM13_CH1			DCMI_PIXCK	EVENTOUT
PA7			TIM1_CHIN	TIM2_CH2	TIM8_CHIN		SPI1_MOSI					TIM14_CH1		ETH_MII_RXDV ETH_RMII_CRS_DV		EVENTOUT

Per tant, el microcontrolador s'hauria de programar per fer servir aquesta funció alternativa en lloc de GPIO pel pin PA5. De la mateixa manera, els altres GPIO associats al perifèric SPI1, PA6 i PA7 també s'haurien de programar amb la funció alternativa AF5 per tenir accés a SPI1\_MISO i SPI1\_MOSI respectivament.

En el nostre cas, però, no cal fer la programació de les funcions alternatives d'aquests pins. La crida que es fa al principi del programa a **baseInit()**, fent servir el fitxer *board.c*, s'encarrega d'inicialitzar tots els perifèrics de la placa STM32F4Discovery. Això inclou no només la inicialització dels ports associats als LEDs que es troben a la placa STM32F4Discovery, sinó que també s'encarrega d'inicialitzar les funcions alternatives dels ports PA5, PA6 i PA7 per què el perifèric SPI1 sigui operatiu ja que aquestes línies estan associades a l'acceleròmetre inclòs a la placa. Finalment també es configuren les línies PE0 i PE1 en mode entrada flotant i la línia PE3 en mode sortida *Push-Pull* amb valor de sortida "1".

## Funcionament de l'acceleròmetre

L'acceleròmetre es comporta, des del punt de vista del microcontrolador, com un conjunt de registres de 8 bits accessibles al bus SPI1. La manera més fàcil de treballar amb el dispositiu és llegir periòdicament els tres registres que mostren les acceleracions dels tres eixos. La següent figura mostra els primers registres del dispositiu. Hi ha 15 registres més, però no ens cal treballar amb ells en aquesta pràctica.

Name	Type	Register address		Default	Comment
		Hex	Binary		
Reserved (Do not modify)		00-0E			Reserved
Who_Am_I	r	0F	000 1111	00111011	Dummy register
Reserved (Do not modify)		10-1F			Reserved
Ctrl_Reg1	rw	20	010 0000	00000111	
Ctrl_Reg2	rw	21	010 0001	00000000	
Ctrl_Reg3	rw	22	010 0010	00000000	
HP_filter_reset	r	23	010 0011	dummy	Dummy register
Reserved (Do not modify)		24-26			Reserved
Status_Reg	r	27	010 0111	00000000	
--	r	28	010 1000		Not Used
OutX	r	29	010 1001	output	
--	r	2A	010 1010		Not Used
OutY	r	2B	010 1011	output	
--	r	2C	010 1100		Not Used
OutZ	r	2D	010 1101	output	

Els registres més importants son els 0x29, 0x2B i 0x2D (de només lectura) que contenen els valors de les acceleracions en els eixos X, Y i Z respectivament.

A part d'aquests registres ens caldrà també treballar amb els **0x0F Who\_Am\_I** i **0x20 Ctrl\_Reg1**.

### **Registre 0x0F : Who\_Am\_I**

Aquest registre, de fet, no és necessari per treballar amb l'acceleròmetre, però ens permetrà comprovar el correcte funcionament del bus SPI.

#### **WHO\_AM\_I (0Fh)**

Table 17. WHO\_AM\_I (0Fh) register

0	0	1	1	1	0	1	1
---	---	---	---	---	---	---	---

Device identification register. This register contains the device identifier that for LIS302DL is set to 3Bh.

El registre permet identificar el dispositiu dins del bus SPI. La lectura d'aquest registre sempre retorna el valor 0x3B.

## Registre 0x20 : Ctr\_Reg1

Aquest registre és important especialment pel bit 6 PD (*Power Down Control*). Quan arrenca, el dispositiu es troba en mode apagat (*Power down mode*). Hem de posar aquest bit PD a “1” per entrar en mode actiu i poder llegir l’acceleració.

### 7.2 CTRL\_REG1 (20h)

Table 18. CTRL\_REG1 (20h) register

DR	PD	FS	STP	STM	Zen	Yen	Xen
----	----	----	-----	-----	-----	-----	-----

Table 19. CTRL\_REG1 (20h) register description

DR	Data rate selection. Default value: 0 (0: 100 Hz output data rate; 1: 400 Hz output data rate)
PD	Power Down Control. Default value: 0 (0: power down mode; 1: active mode)
FS	Full Scale selection. Default value: 0 (refer to <a href="#">Table 3</a> for typical full scale value)
STP, STM	Self Test Enable. Default value: 0 (0: normal mode; 1: self test P, M enabled)
Zen	Z axis enable. Default value: 1 (0: Z axis disabled; 1: Z axis enabled)
Yen	Y axis enable. Default value: 1 (0: Y axis disabled; 1: Y axis enabled)
Xen	X axis enable. Default value: 1 (0: X axis disabled; 1: X axis enabled)

PD bit allows to turn on the turn the device out of power-down mode. The device is in power-down mode when PD= “0” (default value after boot). The device is in normal mode when PD is set to 1.

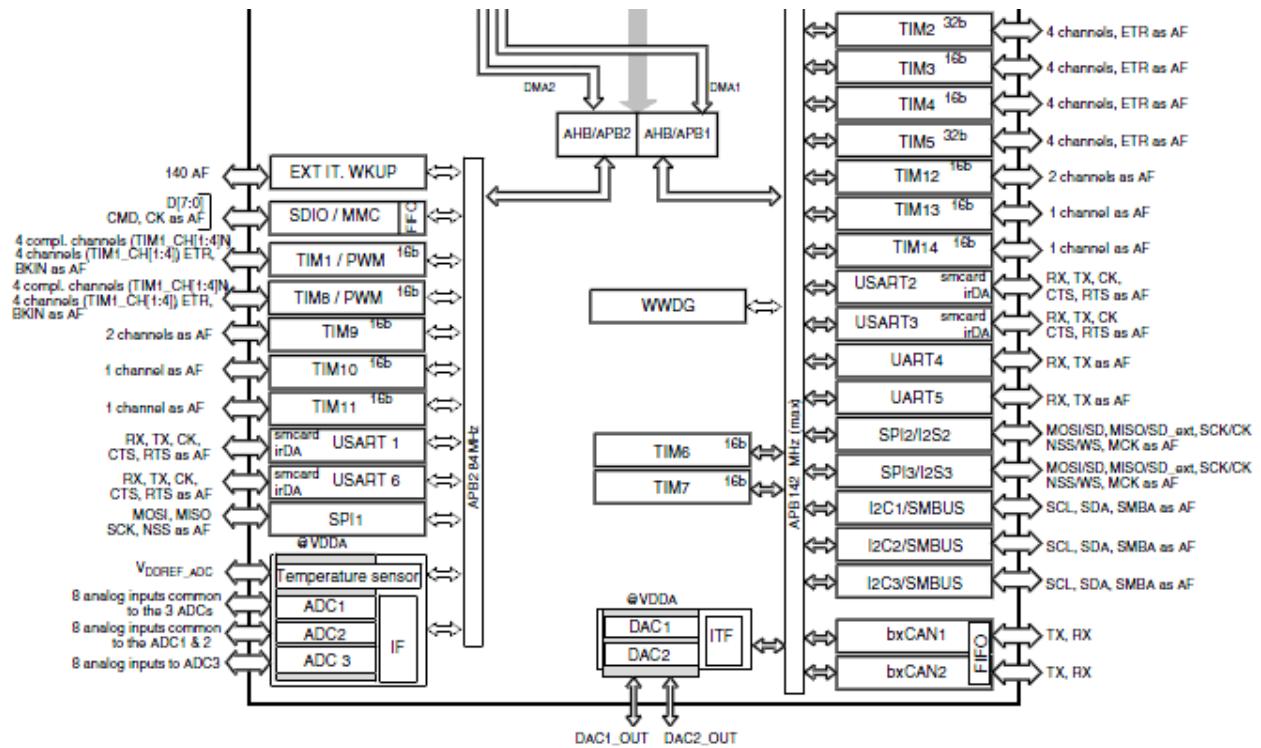
El nostre programa haurà de posar el bit PD del registre 0x20 a “1” per posar en marxa l’acceleròmetre. Vigileu que els bits Xen, Yen i Zen han de mantenir-se al valor per defecte de “1”. A continuació podrem llegir l’acceleració mitjançant els registres X, Y i Z.

La part mes complexa d’aquesta pràctica, però, és fer d’interfície amb el bus SPI.

### P3.2Descripció del bus SPI

El microcontrolador STM32F407 disposa de 3 perifèrics SPI. L'acceleròmetre està connectat al primer d'ells, SPI1.

La figura següent, extreta de la pàgina 17 del Datasheet del microcontrolador STM32F407 mostra els principals perifèrics d'aquest microcontrolador.



El microcontrolador té una estructura complexa de busos interns. En lloc de tenir un únic bus de direccions/dades, tenim múltiples busos. Això permet tenir varíes transferències de dades al mateix temps en cadascun del busos. El bus principal és el bus AHB (*Arm High speed Bus*) que connecta els elements d'alta velocitat com la CPU i la memòria. Aquest bus opera a la freqüència interna de rellotge de sistema de 168 MHz.

A la major part de perifèrics no els hi cal treballar a una freqüència tan alta, iés per això que tenim dos busos perifèrics APB1 (*Arm Peripheral Bus 1*), que opera a 42MHz i APB2 (*Arm Peripheral Bus 2*), que opera a 84MHz. Aquests busos es connecten al bus AHB mitjançant dos ponts *AHB/APB1 Bridge* i *AHB/APB2 Bridge*.

- ① Les freqüències dels rellotges i busos del sistema es poden configurar amb els valors que es defineixen al fitxer **mcuconf.h**

Observem que el perifèric SPI1 penja del bus APB2 que, en el nostre cas, opera a 84 MHz.

Una de les funcionalitats que té el microcontrolador és que els perifèrics que no fem servir, els podem apagar per reduir el consum. Això es fa apagant el senyal de rellotge que els hi arriba. En concret, si no l'activem explícitament, el perifèric SPI1 no tindrà entrada de rellotge i no atendrà a les nostres ordres. Així doncs, el primer que hem de fer és activar el rellotge del perifèric.

El control dels rellotges del microcontrolador el porta el bloc *Reset and clock control* (RCC). La seva descripció es troba a la pàgina 82 del document “STM32F4 Reference.pdf”. Totes el registres del bloc RCC són accessibles amb una estructura **RCC\_TypeDef** descrita dins del fitxer ChibiOS\_2.6.2\os\hal\platforms\STM32F4xx\stm32f4xx.h. El punter **RCC** apunta a aquesta estructura i per tant els seus registres s'adrecen amb el format **RCC->REGISTRE**.

El registre que ens interessa per activar el perifèric SPI1 és APB2ENR accessible al nostre programa com la variable **RCC->APB2ENR**

### 5.3.14 RCC APB2 peripheral clock enable register (RCC\_APB2ENR)

Address offset: 0x44

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
												TIM11 EN	TIM10 EN	TIM9 EN	
												rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reser- ved	SYSCF GEN	Reser- ved	SPI1 EN	SDIO EN	ADC3 EN	ADC2 EN	ADC1 EN	Reser- ved	USART6 EN	USART1 EN	Reser- ved	TIM8 EN	TIM1 EN	rw	rw
			rw	rw	rw	rw	rw		rw	rw				rw	rw

Aquest registre de 32 bits permet controlar el rellotge de 13 perifèrics. El perifèric SPI1 correspon a bit 12 **SPI1EN**. Per activar el perifèric SPI1 hem de posar a “1” aquest bit sense modificar els altres. Tot i que podem accedir al bit 12 ( $2^{12}$ ) fent servir les macros BIT(12) o BIT12 es millor no fer servir constants numèriques al codi. Cada bit del registre RCC\_APB2ENR es pot accedir pel seu nom. En el nostre cas es **RCC\_APB2ENR\_SPI1EN**. Teniu en compte que aquesta constant es  $2^{12}$ , no 12. Les constats associades als registres del microcontrolador es troben al fitxer:

C:\STM32v2\ChibiOS\_2.6.2\os\hal\platforms\STM32F4xx\stm32f4xx.h

En concret, a partir de la línia 6581:

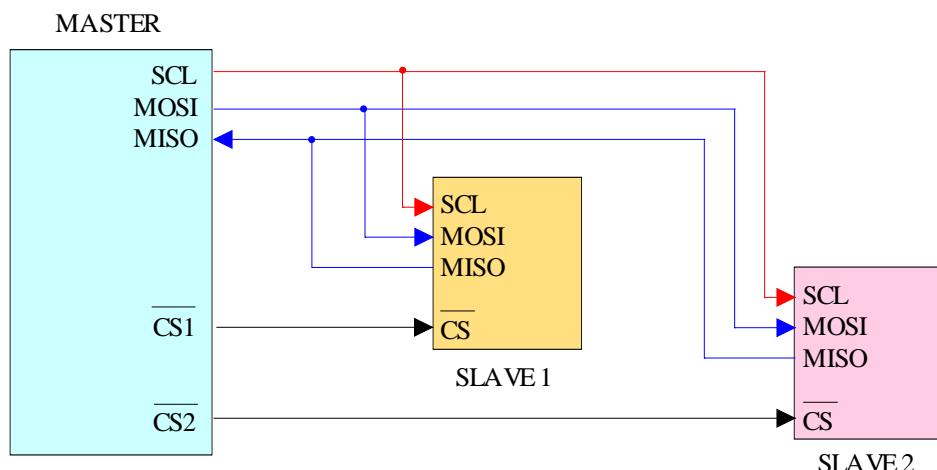
```
/************************************************************************** Bit definition for RCC_APB2ENR register *****/
#define RCC_APB2ENR_TIM1EN ((uint32_t)0x00000001)
#define RCC_APB2ENR_TIM8EN ((uint32_t)0x00000002)
#define RCC_APB2ENR_USART1EN ((uint32_t)0x00000010)
#define RCC_APB2ENR_USART6EN ((uint32_t)0x00000020)
#define RCC_APB2ENR_ADC1EN ((uint32_t)0x00000100)
#define RCC_APB2ENR_ADC2EN ((uint32_t)0x00000200)
#define RCC_APB2ENR_ADC3EN ((uint32_t)0x00000400)
#define RCC_APB2ENR_SDIOEN ((uint32_t)0x00000800)
#define RCC_APB2ENR_SPI1EN ((uint32_t)0x00001000)
```

#define RCC_APB2ENR_SPI4EN	((uint32_t)0x00002000)
#define RCC_APB2ENR_SYSCFGEN	((uint32_t)0x00004000)
#define RCC_APB2ENR_TIM9EN	((uint32_t)0x00010000)
#define RCC_APB2ENR_TIM10EN	((uint32_t)0x00020000)
#define RCC_APB2ENR_TIM11EN	((uint32_t)0x00040000)
#define RCC_APB2ENR_SPI5EN	((uint32_t)0x00100000)
#define RCC_APB2ENR_SPI6EN	((uint32_t)0x00200000)
#define RCC_APB2ENR_SAI1EN	((uint32_t)0x00400000)
#define RCC_APB2ENR_LTDCEN	((uint32_t)0x04000000)

A continuació passarem a descriure el bus SPI. Aquest bus, i la seva programació, estan descrits a partir de la pàgina 658 del document “[STM32F4 Reference.pdf](#)”. El bus SPI és un bus sèrie síncron full dúplex. És sèrie per que les dades es transmeten bit a bit, és síncron per que es transmet el rellotge que cal per recuperar les dades i és full dúplex per que es pot transmetre i rebre dades al mateix temps.

De manera similar al bloc **RCC**, totes el registres del bloc SPI1 són accessibles amb una estructura **SPI\_TypeDef** descrita dins del fitxer *stm32f4xx.h* descrit anteriorment. El punter **SPI1** apunta a aquesta estructura i per tant tots els seus registres s’adrecen amb el format **SPI1->REGISTRE**.

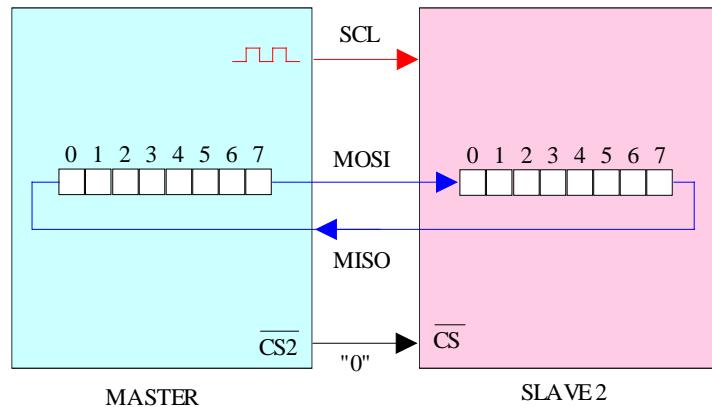
En un bus SPI tenim un dispositiu *master* i un conjunt de dispositius *slave*. En el nostre cas el microcontrolador STM32F407 actuarà com a *master* i l’acceleròmetre LIS302DL actuarà com a *slave*. Els perifèrics SPI del microcontrolador poden operar també com a *slave*, però aquesta funcionalitat no la farem servir. La següent figura mostra una imatge d’un exemple de connexió SPI entre un *master* i dos *slave*.



En un moment donat només un *slave* pot comunicar-se amb el *master*. La selecció la fa el dispositiu *master* mitjançant senyals de *chip select* actives per nivell baix. Cada *slave* té una entrada nCS (CS activa baixa) i, en tot moment, només un *slave* pot tenir activa la seva entrada nCS. Aquest és el dispositiu que es comunicarà amb el *master*. Els perifèrics SPI del microcontrolador poden gestionar automàticament la generació dels senyals nCS. En el nostre cas, no obstant, gestionarem els senyals nCS manualment fent servir línies GPIO normals. Recordem que el senyal CS de l’acceleròmetre està connectat a **PE3** per tant, aquesta línia la haurem de configurar com a GPIO de sortida Push-Pull.

Com que la comunicació només pot fer-se, en un moment donat, amb un *slave*, continuarem la descripció funcional del bus parlant només d'un *master* i un *slave*.

Les dades es transmeten mitjançant dues línies MOSI (PA7) i MISO (PA6). MOSI vol dir **Master Output / Slave Input**, és a dir, comunicació de *master* cap a *slave*. MISO vol dir **Master Input / Slave Output**, és a dir, comunicació de *slave* cap a *master*. La comunicació es fa normalment amb blocs d'un Byte (8 bits). La següent figura mostra la comunicació entre el *master* i el segon *slave*.



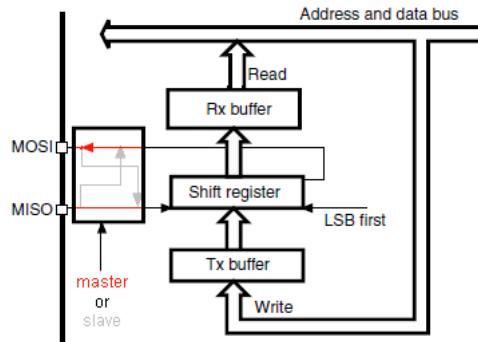
Abans de la transmissió, tots dos dispositius tenen una dada de 8 bits a transmetre. Al primer cicle de rellotge es fa un desplaçament del registre del *master* traient per **MOSI** el bit 7. Aquest bit entra al bit 0 del registre del *slave* desplaçant els altres bits. El bit 7 del *slave* surt per **MISO** i passa a ocupar el bit 0 del *master*. El procés es repeteix fins que es completen 8 cicles de rellotge moment en el qual *master* i *slave* han intercanviat la informació dels seus registres.

## Registre DR

El registre de dades del microcontrolador es diu **SPI data register (SPI\_DR)** i és accessible com SPI1->DR.

25.5.4 SPI data register (SPI_DR)																
Address offset: 0x0C																
Reset value: 0x0000																
DR[15:0]																
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
Bits 15:0 DR[15:0]: Data register Data received or to be transmitted. The data register is split into 2 buffers - one for writing (Transmit Buffer) and another one for reading (Receive buffer). A write to the data register will write into the Tx buffer and a read from the data register will return the value held in the Rx buffer.																
Notes for the SPI mode: <i>Depending on the data frame format selection bit (DFF in SPI_CR1 register), the data sent or received is either 8-bit or 16-bit. This selection has to be made before enabling the SPI to ensure correct operation.</i> <i>For an 8-bit data frame, the buffers are 8-bit and only the LSB of the register (SPI_DR[7:0]) is used for transmission/reception. When in reception mode, the MSB of the register (SPI_DR[15:8]) is forced to 0.</i> <i>For a 16-bit data frame, the buffers are 16-bit and the entire register, SPI_DR[15:0] is used for transmission/reception.</i>																

El registre **DR**, de fet, tal i com s'indica al quadre anterior, no es el registre de desplaçament que fa la comunicació sinó un accés a dos *buffers* independents, un de transmissió i un altre de recepció. La següent figura obtinguda de “[STM32F4 Reference.pdf](#)” mostra aquest comportament:



Les dades s'escriuen en el registre de desplaçament mitjançant un *buffer* de transmissió i es llegeixen del registre de desplaçament mitjançant un *buffer* de recepció. Però, tots dos *buffers* s'accedeixen mitjançant el registre DR. Per tant, per enviar i rebre una dada pel bus SPI amb el microcontrolador fent de *master* farem:

```
SPI1->DR = Dada; // Write data to be sent
..... // Wait to end transmission
Dada = SPI1->DR; // Read received data
```

Com es veu, la lectura i escriptura es fan com si no hi haguessin *buffers*. Els *buffers*, però, són necessaris per permetre llegir l'última dada rebuda quan s'està transmetent la següent i per escriure la paraula que s'ha de transmetre just quan acaba la transmissió actual. Per tant, una vegada comença la transmissió d'un Byte, ja es pot escriure el següent al registre DR i serà enviat tan bon punt s'hagi acabat la transmissió del anterior.

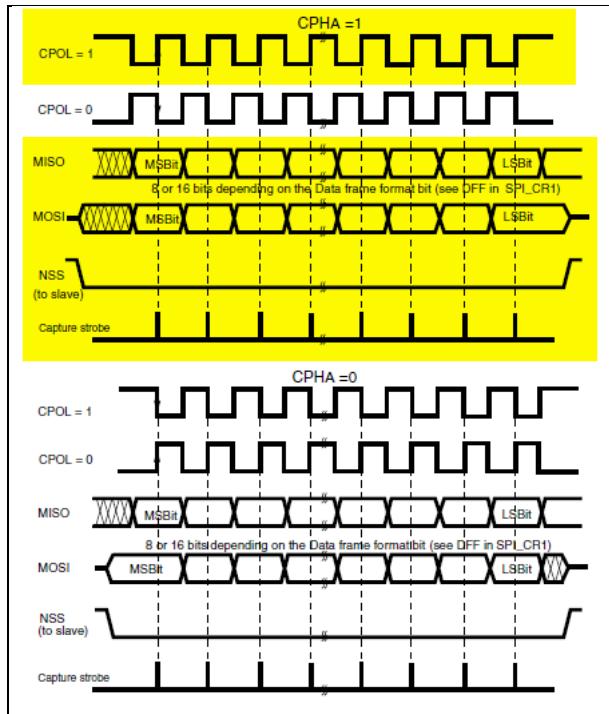
## Rellotge SCL

La definició del senyal de rellotge SCL (**PA5**) al bus SPI no és universal ja que n'hi han 4 maneres diferents de fer servir el rellotge SCLK. El control del rellotge es fa mitjançant 2 dades binàries **CPOL** i **CPHA**.

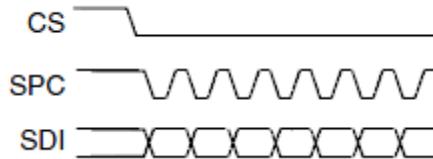
**CPOL** vol dir *Clock Polarity*, es a dir, signe del senyal de rellotge. Aquesta dada indica el valor que ha de tenir el senyal de rellotge quan no es transmeten dades (línia inactiva). Si CPOL val “0”, la línia es inactiva per nivell baix i si val “1” la línia es inactiva per nivell alt.

**CPHA** vol dir *Clock Phase*. Aquesta dada indica quan s'han de llegir les dades. Si CPHA val “0”, la primera dada s'ha de llegir en el primer flanc del rellotge des de l'estat de repòs definit amb CPOL. Si CPHA val “1” la primera dada s'ha de llegir en el segon flanc del rellotge, es a dir, quan el rellotge torna al estat de repòs definit per CPOL.

La següent figura mostra les 4 possibles combinacions de CPOL i CPHA.



S'ha marcat en groc la combinació CPOL=CPHA="1" per que és la que cal fer servir amb l'acceleròmetre LIS302DL segons es dedueix de la figura següent obtinguda de la pàgina 22 del document "LIS302DL.pdf".



El control de CPOL i CPHA es fa mitjançant els bits 0 i 1 del registre **SPI control register 1** (CR1) accessible com SPI1->CR1

#### 25.5.1 SPI control register 1 (SPI\_CR1) (not used in I<sup>2</sup>S mode)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRC EN	CRC NEXT	DFF	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR [2:0]			MSTR	CPOL	CPHA
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Recordeu que els bits CPOL y CPHA es poden accedir amb els noms SPI\_CR1\_CPOL i SPI\_CR1\_CPHA i que aquesta solució es millor que fer servir BIT(1) i BIT(0).

## Altres configuracions

Per poder treballar amb el bus SPI, a part de la configuració del rellotge mitjançant CPOL i CPHA, també cal fer altres configuracions.

Per començar cal configurar la velocitat de transmissió. Com s'ha indicat, el perifèric SPI1 penja del bus APB2 que opera a 84 MHz. Els tres bits 5, 4 i 3 del registre **SPI control register 1** (CR1) controlen com es divideix el rellotge APB2 per obtenir el rellotge SCL. És possible configurar divisions des de  $f_{APB2}/2$  fins a  $f_{APB2}/256$ .

### Bits 5:3 BR[2:0]: Baud rate control

000: $f_{PCLK}/2$	100: $f_{PCLK}/32$
001: $f_{PCLK}/4$	101: $f_{PCLK}/64$
010: $f_{PCLK}/8$	110: $f_{PCLK}/128$
011: $f_{PCLK}/16$	111: $f_{PCLK}/256$

*Note: These bits should not be changed when communication is ongoing.*

*Not used in PS mode*

El camp BR té un nom amb una sintaxi different, per exemple, de CPOL i CPHA. Això es degut a que té més d'un bit associat. Observem les definicions que tenim per aquests camps al fitxer **stm32f4xx.h**:

```
#define SPI_CR1_BR ((uint16_t)0x0038) /*!<BR[2:0] bits (Baud Rate Control) */
#define SPI_CR1_BR_0 ((uint16_t)0x0008) /*!<Bit 0 */
#define SPI_CR1_BR_1 ((uint16_t)0x0010) /*!<Bit 1 */
#define SPI_CR1_BR_2 ((uint16_t)0x0020) /*!<Bit 2 */
```

Els noms SPI\_CR1\_BR\_0 a SPI\_CR1\_BR2 corresponen a cadascun dels bits 0 a 2 del camp BR de 3 bits. El nom SPI\_CR\_BR correspón a la combinació OR dels tres nombres anteriors (equivalent a tenir els tres bits a 1). Per possar un nombre n entre 0 i 15 a la posició que ocupa BR podriem fer servir:

$$n \ll 3 \quad \text{que es equivalent a} \quad n * (1 \ll 3)$$

Però, com que no volem fer servir constants numèriques, podem fer servir en el seu lloc:

$$n * \text{SPI\_CR1\_BR0}$$

Per cambiar BR amb el valor n sense tocar la resta del registre CR1 podem fer servir:

$$\text{SPI1\<-CR1} = ((\text{SPI1\<-CR1}) \& (\sim \text{SPI\_CR1\_BR})) | (n * \text{SPI\_CR1\_BR0})$$

La operació **&** borra el contingut de BR i la operació **|** possa **n** en el seu lloc. Si aquest tipus de manipulació de bits us sembla massa complexa, podeu fer servir qualsevol codi equivalent.

La freqüència màxima de rellotge per l'acceleròmetre treballant com a *slave* al bus SPI es troba al document “LIS302DL.pdf”.

EP1 – Determineu la freqüència màxima que es pot programar a BR[2:0] que sigui compatible amb l'acceleròmetre. Determineu els valor que cal configurar a BR[2:0].

També s'ha de configurar el bit 11 (SPI\_CR1\_DFF) en format de 8 bits.

**Bit 11 DFF: Data frame format**

0: 8-bit data frame format is selected for transmission/reception

1: 16-bit data frame format is selected for transmission/reception

*Note: This bit should be written only when SPI is disabled (SPE = '0') for correct operation  
Not used in PS mode*

Donat que treballem amb el microcontrolador actuant com a *master*, haurem de configurar adequadament el bit 2 (SPI\_CR1\_MSTR).

**Bit 2 MSTR: Master selection**

0: Slave configuration

1: Master configuration

*Note: This bit should not be changed when communication is ongoing.  
Not used in PS mode*

Els bits 9 (SPI\_CR1\_SSM) i 8 (SPI\_CR1\_SSI) configuren l'ús de nCS en mode de entrada. Nosaltres gestionarem els senyals nCS per software fent servir pins GPIO. Per aquest motiu tots dos bits s'han de posar a “1”.

**Bit 9 SSM: Software slave management**

When the SSM bit is set, the NSS pin input is replaced with the value from the SSI bit.

0: Software slave management disabled

1: Software slave management enabled

*Note: Not used in PS mode and SPI TI mode*

**Bit 8 SSI: Internal slave select**

This bit has an effect only when the SSM bit is set. The value of this bit is forced onto the NSS pin and the IO value of the NSS pin is ignored.

*Note: Not used in PS mode and SPI TI mode*

El bit 6 (SPI\_CR1\_SPE) habilita el perifèric SPI. Per posar en marxa el perifèric, s'ha d'activar aquest bit. Els bits 6 (SPE) i 2 (MSTR) son els últims que s'han de programar.

**Bit 6 SPE: SPI enable**

0: Peripheral disabled

1: Peripheral enabled

*Note: 1- Not used in PS mode.*

*Note: 2- When disabling the SPI, follow the procedure described in Section 25.3.8: Disabling the SPI.*

## Registre d'estat

El perifèric SPI disposa d'un registre d'estat *SPI status register* (SR) accessible com SPI->SR

D'aquest registre interessa especialment el bit 0 (SPI\_SR\_RXNE) ja que aquest s'activa quan s'ha omplert el *buffer* de recepció. Es a dir quan ha acabat la transmissió en curs. Aquest bit, per tant, ens permetrà saber quan ha acabat l'últim enviament.

### 25.5.3 SPI status register (SPI\_SR)

Address offset: 0x08

Reset value: 0x0002

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						TIFRFE	BSY	OVR	MODF	CRC ERR	UDR	CHSID E	TXE	RXNE	
						r	r	r	r	rc_w0	r	r	r	r	

#### Bit 0 RXNE: Receive buffer not empty

0: Rx buffer empty

1: Rx buffer not empty

## P3.3Configuració del bus SPI

Per configurar el bus SPI hem de realitzar les següents accions:

- Habilitar el rellotge del perifèric SPI1 del bus APB2
- Configurar el registre CR1
  - Configurar CPOL i CPHA
  - Configurar DFF
  - Configurar BR[2:0] amb el valor obtingut a EP1
  - Configurar SSM i SSI
- Configurar el pin PE3 que fa de nCS com GPIO de sortida *Push-Pull* i posar-lo a nivell alt (*slave* no seleccionat). Aquest punt, de fet, es opcional perquè el sistema arrenca després de *baseInit( )* en aquest estat.
- Activar SPE i MSTR al registre CR1

Recordeu que no cal seleccionar les funcions alternatives de PA5, PA6 i PA7 ja que es fa dintre de baseInit( ). De fet, la inicialització de nCS es fa també en aquesta funció.

Recordeu també que no es bona pràctica fer servir constants numèriques quan accedim als registres. Feu servir els noms definits a `stm32f4xx.h`.

EP2 Escriviu una funció `void initAccel(void)` que realitzi les operacions anteriors.

### Inici del treball de laboratori

En aquest punt s'inicia el treball de laboratori. Això no vol dir que no hi hagi més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

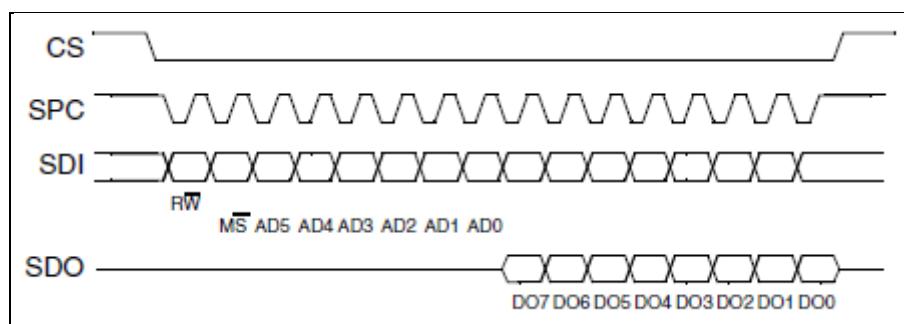
⚡ Abans que res, comproveu el bon funcionament de la placa.

Per fer aquesta pràctica disposeu dels fitxers **accel.c** i **accel.h** continguts al fitxer **Practica-P3.zip**.

Afegeiu aquests fitxers al vostre directori **practica** i incorporeu-los al projecte modificant el fitxer *makefile*. Feu servir el mateix procediment que es fa servir a la pràctica P2 pels fitxers **lcd.c** i **lcd.h**

### P3.4 Lectura de dades de l'acceleròmetre

La descripció de l'acceleròmetre en mode de lectura es troba a la pàgina 22 del document “LIS302DL.pdf”.



Les dades es transmeten pel bus SPI començant pel bit de major pes (MSB). Per tant, el primer bit d'un byte que es transmet es el bit 7 (MSB).

Per llegir un registre de l'acceleròmetre s'han d'enviar dos bytes:

El primer byte indica quin registre volem llegir i conté la següent informació:

- Bit 7 (RW) Per indicar que volem llegir un registre s'ha de posar a “1”
- Bit 6 (MS) Autoincrement. El posarem a “0” (veure explicació a baix)
- Bits 5..0 (AD) Nombre del registre que es vol llegir

El bit 6 (MS) permet llegir N registres en N+1 transferències en lloc de fer servir 2N transferències (Direcció+Dada). Activant l'autoincrement, la direcció a llegir s'incrementa amb cada lectura de manera que podem llegir més dades de registres consecutius sense donar la direcció de cada registre.

En el nostre cas només volem llegir un registre i, per tant, no cal fer servir l'autoincrement.

El segon byte enviat es fa servir per poder rebre la resposta de l'acceleròmetre.

El bus SPI es full dúplex, però, amb el protocol indicat a dalt, la comunicació es símplex. El primer Byte es només de *master* cap a *slave* i el segon es només de *slave* cap a *master*. Com que la comunicació es intrínsecament dúplex, per fer una escriptura símplex, s'ha de descartar el contingut del *buffer* de recepció, i, per fer una lectura, s'ha de fer servir una dada *dummy* arbitrària de transmissió.

Tot això es pot veure a la figura anterior. El primer byte comença amb l'enviament de RW i acaba amb l'enviament d'AD0. Durant el temps que es fa aquest enviament, l'acceleròmetre està enviant cap informació, per tant, descartem la informació del buffer de recepció.

El segon byte és transmissió del acceleròmetre cap al microcontrolador. Durant aquest segon byte, la informació que enviem no es llegida (es tracta d'un *dummy byte*) només importa la informació que posa l'acceleròmetre en el bus que anirà al buffer de recepció.

El procediment per llegir un registre és, per tant:

- Activar el senyal nCS posant-lo a “0”
- Escriure un Byte a DR amb el nombre de registre i els bits RW i MS
- Esperar una mica abans de començar a llegir el flag RXNE
- Esperar que s'activi el flag RXNE (dada en *buffer* de recepció)
- Fer una lectura a DR per buidar el *buffer* de recepció
- Escriure una dada qualsevol a DR per començar la recepció de la resposta
- Esperar una mica abans de començar a llegir el flag RXNE
- Esperar que s'activi el flag RXNE (dada en *buffer* de recepció)
- Llegir DR que contindrà el contingut del registre
- Desactivar el senyal nCS posant-lo a “1”

Tot aquest procediment està a la funció **readAccel** que es troba dins del fitxer **accel.c**

Per facilitar la presentació de dades al LCD, us proporcionem la funció **itoa** que permet convertir un nombre en una cadena dins del fitxer **accel.c**

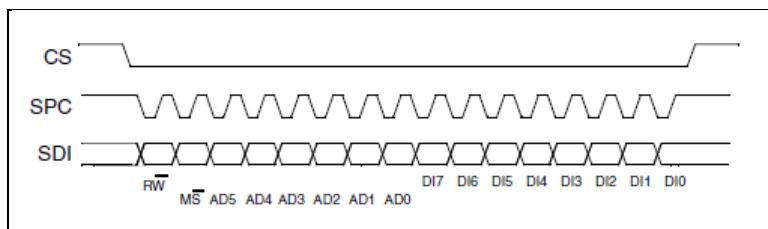
- Incloeu la funció **initAccel** desenvolupada dins de EP2 dins d'**accel.c**.
- Modifiqueu la funció **main** per què es cridi a **baseInit**, **LCD\_Init** i **initAccel**. Feu que a continuació es llegeixi el registre **Who\_Am\_I** cridant la funció **readAccel** que us donem. Feu que es mostri el resultat al LCD fent servir la funció **itoa**. El valor llegit ha de ser 0x3B.

### P3.5Escriptura de dades al'acceleròmetre

Si intentem llegir ara alguns del registres de sortida de l'acceleròmetre X, Y o Z, obtindrem un valor de 0. Això és degut a que, per defecte, l'acceleròmetre es troba en mode power down. Per activar-lo hem d'escriure al registre 0x20 *Ctr\_Reg1*. Aquest registre ja ha estat descrit abans, a P3.1.

- Vigileu per que *Ctrl\_Reg1*, a part del bit Power Down, té també bits que habiliten els tres eixos X, Y, Z. No els esborreu per error quan activeu l'acceleròmetre.

El protocol per escriure a un registre de l'acceleròmetre és similar al que hem fet servir per llegir-lo. De fet, la primera part, en la que s'envia un Byte amb l'adreça del registre és igual excepte pel fet que el bit 7 (RW) l'hem de posar a "0" per indicar que es una escriptura. En el segon Byte que enviem, en lloc d'enviar una dada *dummy* arbitraria per poder rebre la resposta, enviarem el contingut que volem escriure al registre.



Com que només hem d'escriure al'acceleròmetre per activar-lo, no cal, si no voleu, que definiu una funció **writeAccel**. El codi per escriure al registre podeu incloure'l a la funció **initAccel**.

EP3 Escriviu una primera versió del codi que s'ha d'incloure al final d'**initAccel** per posar en marxa l'acceleròmetre.

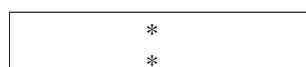
- Modifiqueu la vostra funció **initAccel** per què, després d'inicialitzar el bus SPI 1, s'activi l'acceleròmetre accedint al registre *Ctrl\_Reg1*.

- ¶ Podeu fer el codi mes genèric si definiu una funció **writeAccel** tal i com es descriu més amunt per escriure als registres de l'acceleròmetre i, en particular, a *Ctrl\_Reg1*. Si observeu com està definida la funció **readAccel** que us donem, veureu que retorna un codi d'error quan s'intenta accedir a un registre reservat. En el cas de fer una funció d'escriptura això és especialment important ja que, tal i com es diu a la pàgina 25 del document “LIS302DL.pdf”, l'escriptura als registres reservats pot causar un dany permanent al dispositiu. A més a més, també s'ha de generar error quan s'intenta escriure en un registre de només lectura.
- Modifiqueu **main** per què s'inicialitzi la placa, l'LCD i l'acceleròmetre i es mostri periòdicament el valor del registre Y de l'acceleròmetre.  
 Els registres X, Y i Z son Bytes amb signe, per tant, el seu rang màxim es -128 a 127.  
 Verifiqueu que els valor canvien en inclinar la placa de costat.

### **P3.6Programa final**

Per acabar aquesta pràctica farem un programa que mostri en pantalla dos caràcters "\*", un a la primera filera i un altre a la segona.

Quan engega el programa ha de llegir les acceleracions X i Y i considerar-les com a posició de zero. A partir d'aquest moment, en un bucle repetitiu, han de canviar les posicions dels caràcters de cada eix segons sigui la inclinació de la placa en els eixos X (primera filera) i Y (segona filera).



Imatge quan arrenca



Eixos X i Y en direccions oposades

Els màxims d'inclinació haurien de ser les corresponents als valors associats a unes inclinacions de la placa al voltant de  $\pm 45^\circ$ .

Per determinar els valors que corresponen a aquesta inclinació, podeu obtenir-los de la sensibilitat del sensor que és, per defecte, de 0.018 g/dígit (on g es la gravetat de la terra), o obtenir-los, experimentalment, amb el programa del apartat P3.5 que mostra els valor numèrics dels registre Y.

- ¶ Els caràcters '\*' en cap cas han de sortir de la pantalla, tots dos han de ser sempre visibles.
- ¶ Per evitar fluctuacions degudes al soroll, podeu fer servir una matriu per emmagatzemar les N últimes lectures i treballar amb el valor mig d'aquestes. El valor N s'ha d'escollir com a compromís entre soroll i velocitat de resposta.

### **EP4 Escriviu una primera versió del codi del programa demanat.**

- Desenvolupeu i carregueu el programa.
- ↳ Proveu el funcionament de la placa inclinant-la en els eixos X i Y

Amb això acaba la pràctica P3

## P4 – Interrupcions i mesures de temporització

En el punt actual de les pràctiques podem desenvolupar i depurar pas a pas i mitjançant *breakpoints* els programes que fem pel microcontrolador STM32F407. Per fer una anàlisis completa del funcionament d'un sistema basat en microprocessador no n'hi ha prou amb aquestes eines, s'han de poder fer també mesures en els senyals digitals. Això vol dir treballar amb l'analitzador lògic.

A més a més, els programes de les pràctiques P1 a P3 son programes senzills que només tenen una part d'inicialització i un bucle infinit. Normalment els sistemes amb microprocessador han de poder processar eventualitats externes fent servir interrupcions. En aquesta pràctica en farem una petita introducció.

### Objectius de la pràctica:

- Aprendre a treballar amb interrupcions associades als ports de GPIO.
- Ser capaç de mesurar temps de resposta d'interrupcions
- Verificar les temporitzacions del bus SPI

### Estudis Previs:

- EP1 Codi de la funció demanada a P4.2
- EP2 Codi de la RSI del polsador demanat a P4.2

### Resultats:

- Funcionament del programa demanat a P4.2
- Realització de les mesures demandades a P4.3

### Resultats opcionals:

- No n'hi ha cap de proposat

### P4.1 Gestió d'interrupcions als ports GPIO

En sistemes que interactuen amb el món exterior es poden generar eventualitats que han de ser ateses a temps. Normalment aquestes eventualitats estan associades a la generació d'interrupcions que el processador ha d'atendre. Tots els processadors ARM-Cortex, com és el cas del nostre, tenen un element de hardware específic per tractar amb les interrupcions que es diu *Nested vectored interrupt controller* (NVIC).

Gairebé tots els perifèrics del microcontrolador STM32F07 son capaços de generar interrupcions. Treballar amb interrupcions és eficient per que podem oblidar-nos del perifèric fins que a aquest li calgui l'atenció del processador, moment en el qual genera una interrupció.

La interrupció, si té prou prioritat, farà que el processador deixi de fer el que estava fent per dedicar-se a atendre el perifèric. A nivell pràctic, la generació d'un senyal d'interrupció fa que es cridi una rutina de servei d'interrupció (RSI) de manera asíncrona al funcionament de la resta del programa. Quan aquesta funció acaba, el programa principal continua amb el que estava fent.

La gestió d'interrupcions a les línies de GPIO involucra la programació de tres elements diferents. En primer lloc hem d'indicar quines línies de GPIO, de les 82 disponibles, poden generar interrupcions. Això es fa amb el *System configuration controller* (SYSCFG). En segon lloc s'ha de programar en quines condicions aquestes línies poden generar interrupcions. Això es fa amb el *External interrupt controller* (EXTI). Finalment, hem de programar el *Nested vectored interrupt controller* (NVIC) per que, quan es generi aquesta interrupció, es doni curs a la crida de la RSI.

La informació sobre la gestió d'interrupcions i, en concret, la gestió d'interrupcions de les línies de GPIO es troba a partir de la **pàgina 195** del document "[STM32F4 Reference.pdf](#)".

És una mica complex, però ho explicarem pas a pas.

### System configuration controller (SYSCFG)

Per poder treballar amb el *System configuration controller* (SYSCFG), com que és, en si mateix, un perifèric, s'ha d'activar el seu rellotge. Aquest perifèric penja del bus perifèric APB2 igual que ho feia el perifèric SPI 1 de la pràctica P3. Per tant, s'activa amb el mateix registre RCC ->APB2ENR

#### 5.3.14 RCC APB2 peripheral clock enable register (RCC\_APB2ENR)

Address offset: 0x44

Reset value: 0x0000 0000

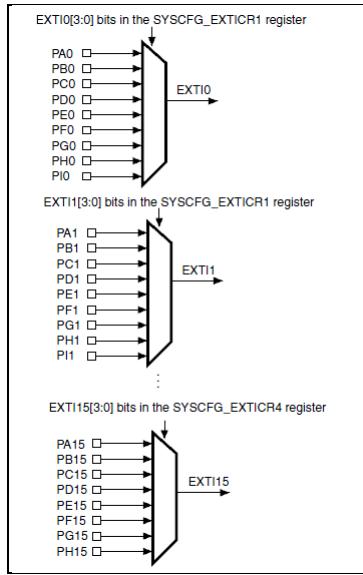
Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reser- ved	SYSCF GEN	Reser- ved	SPI1 EN	SDIO EN	ADC3 EN	ADC2 EN	ADC1 EN	Reserved	USART6 EN	USART1 EN	Reserved	TIM8 EN	TIM1 EN	TIM9 EN	rw

En aquest cas, però, el bit a activar no és el 12 (SPI1EN) sinó el 14 (SYSCFGEN). Recordeu la notació de noms per anomenar aquests bits.

El nostre microcontrolador té 82 línies de GPIO. Per qüestions pràctiques no es possible tenir interrupcions independents associades a totes les 82 línies. La selecció de les línies de GPIO que poden donar lloc a interrupcions es fa amb una part d'un bloc que es diu *System configuration controller* (SYSCFG).

Com ja sabem de la pràctica P1, els pins de GPIO estan agrupats en ports de fins a 16 línies numerades de 0 a 15 per cada port. En total podem fer servir 16 línies de GPIO al mateix temps per generar una interrupció. La selecció de línies que poden generar interrupcions es fa mitjançant un multiplexor tal i com es mostra a la següent figura:



Com es veu, la línia 0 (EXTI 0) pot venir de PA0, PB0, etc.. es a dir, de qualsevol línia 0 de qualsevol port. Igual passa amb les línies 1 a 15 per obtenir les 16 línies de GPIO que poden generar interrupcions. Això vol dir que dos línies de GPIO amb el mateix nombre, PA0 i PB0, per exemple, no poden generar interrupcions al mateix temps.

El control dels multiplexors es fa amb 4 registres del *System configuration controller* (SYSCFG) que es diuen EXTICR1, EXTICR2, EXTICR3 i EXTICR4.

Cada registre controla 4 dels multiplexors fins a un total de 16.

### 7.2.3 SYSCFG external interrupt configuration register 1 (SYSCFG\_EXTICR1)

Address offset: 0x08

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI3[3:0]				EXTI2[3:0]				EXTI1[3:0]				EXTI0[3:0]			
rw	rw	rw	rw												

Els quatre camps EXTI, amb tots els bits a “1”, es referencien amb noms SYSCFG\_EXTICR1\_EXTI0 a SYSCFG\_EXTICR1\_EXTI3 al fitxer *stm32f4xx.h*.

#### 7.2.4 SYSCFG external interrupt configuration register 2 (SYSCFG\_EXTICR2)

Address offset: 0x0C

Reset value: 0x0000

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

#### 7.2.5 SYSCFG external interrupt configuration register 3 (SYSCFG\_EXTICR3)

Address offset: 0x10

Reset value: 0x0000

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

#### 7.2.6 SYSCFG external interrupt configuration register 4 (SYSCFG\_EXTICR4)

Address offset: 0x14

Reset value: 0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI15[3:0]				EXTI14[3:0]				EXTI13[3:0]				EXTI12[3:0]			
rw	rw	rw	rw												

Els quatre camps EXTI del registre EXTICR2, amb tots els bits a “1”, es referencien amb noms SYSCFG\_EXTICR2\_EXTI4 a SYSCFG\_EXTICR2\_EXTI7 al fitxer *stm32f4xx.h*. De la mateixa manera tenim per EXTICR3 tenim de SYSCFG\_EXTICR3\_EXTI8 a SYSCFG\_EXTICR3\_EXTI11 i, finalment, per EXTICR4 tenim noms de constants de SYSCFG\_EXTICR4\_EXTI12 a SYSCFG\_EXTICR4\_EXTI15 .

Com veiem, tenim 4 bits associats a cada línia EXTI0..15, aquests bits indiquen a quin port pertany cada línia EXTI segons la següent distribució de valors:

Valor	PORT
0000	A
0001	B
0010	C
0011	D
0100	E
0111	H

Recordeu que al STM32F407 que fem servir, amb encapsulat de 100 pins, el port H només té 2 línies. En altres versions del microcontrolador hi han ports addicionals fins arribar a 140 línies de GPIO. En aquests casos tenim també ports amb les lletres I i J.

Les constants associades als ports GPIO troben definides també a *stm32f4xx.h*. En concret tenim les possibilitats per cada port A...J per cada camp EXTI 0...15 amb un total de 160 constantes des de **SYSCFG\_EXTICR1\_EXTI0\_PA** fins a **SYSCFG\_EXTICR4\_EXTI15\_PJ**.

Fer servir els noms definits a *stm32f4xx.h* per configurar les línies EXTI es més complex que en altres casos per tant, en cas de dubte, podeu provar de fer servir constants numèriques tot i que es una pitjor solució.

Per programar aquests registres tenim un punter **SYSCFG** que apunta a una estructura **SYSCFG\_TypeDef** que es troba descrita a *stm32f4xx.h*.

```
typedef struct
{
    __IO uint32_t MEMRMP;
    __IO uint32_t PMC;
    __IO uint32_t EXTICR[4];
    uint32_t      RESERVED[2];
    __IO uint32_t CMPCCR;
} SYSCFG_TypeDef;
```

Observeu que els registres EXTICR1 a fins EXTICR4, en ser consecutius, es troben definits mitjançant una matriu EXTICR de 4 membres. Per tant, EXTICR1 es EXTICR[0], EXTICR2 es EXTICR[1] i així successivament.

<b>SYSCFG-&gt;EXTICR[0]</b>	EXTI0 fins a EXTI3
<b>SYSCFG-&gt;EXTICR[1]</b>	EXTI4 fins a EXTI7
<b>SYSCFG-&gt;EXTICR[2]</b>	EXTI8 fins a EXTI11
<b>SYSCFG-&gt;EXTICR[3]</b>	EXTI12 fins a EXTI15

Una vegada seleccionades les 16 línies de GPIO que es poden fer servir per generar interrupcions, s'ha de programar el que cal fer amb elles.

### External interrupt controller (EXTI)

L'*External interrupt controller* (EXTI) és el subsistema del microcontrolador STM32F407 que gestiona les interrupcions associades a pins de GPIO seleccionats mitjançant el *System configuration controller*.

Com sabem, tenim 16 possibles línies GPIO que poden generar interrupcions, però l'*External interrupt controller* (EXTI), en realitat gestiona 23 línies. Les línies 0 a 15 corresponen a les 16 línies escollides amb el *System configuration controller* segons s'ha explicat a l'apartat anterior. La resta son casos especials. La descripció del conjunt de línies s'indica a la taula següent:

Línia	Descripció
0 a 15	Línies GPIO escollides amb el <i>System configuration controller</i>
16	Petició del monitor de tensió d'alimentació (PVD)
17	Alarma del rellotge en temps real (RTC)
18	Petició USB OTG Full Speed
19	Petició de d'interfície Ethernet
20	Petició USB OTG High Speed
21	Esdeveniments del rellotge en temps real (RTC) <i>Tamper i TimeStamp</i>
22	Petició <i>Wakeup</i> del rellotge en temps real (RTC)

Tota la configuració es fa mitjançant els següents registres que son accessibles amb el punter **EXTI**.

### 9.3.1 Interrupt mask register (EXTI\_IMR)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved														MR22	MR21
									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

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **MRx**: Interrupt mask on line x

0: Interrupt request from line x is masked

1: Interrupt request from line x is not masked

El registre *Interrupt mask register(IMR)*, accessible com **EXTI->IMR**, indica, per cadascuna de les 23 fonts d'interrupció, si la interrupció està habilitada o emmascarada. Només les línies que tinguin el seu bit a "1" donaran lloc a peticions d'interrupció.

Els noms dels bits a *stm32f4xx.h* van de **EXTI\_IMR\_MR0** fins a **EXTI\_IMR\_MR19**.

### 9.3.3 Rising trigger selection register (EXTI\_RTSR)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved														TR22	TR21
									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

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **TRx**: Rising trigger event configuration bit of line x

- 0: Rising trigger disabled (for Event and Interrupt) for input line
- 1: Rising trigger enabled (for Event and Interrupt) for input line

### 9.3.4 Falling trigger selection register (EXTI\_FTSR)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved														TR22	TR21
									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

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 **TRx**: Falling trigger event configuration bit of line x

- 0: Falling trigger disabled (for Event and Interrupt) for input line
- 1: Falling trigger enabled (for Event and Interrupt) for input line.

Els registres *Rising trigger selection register (RTSR)* i *Falling trigger selection register (FTSR)* accessibles, respectivament, com **EXTI->RTSR** i **EXTI->FTSR**, permeten determinar si les interrupcions s'han de donar per flanc de pujada, per flanc de baixada o per tots dos flancs.

Per activar una línia per flanc de pujada posarem el seu bit del registre **RTSR** a "1". Igualment, per activar una línia per flanc de baixada posarem el seu registre **FTSR** a "1". Si, per una línia donada, tant el bit del registre **RTSR** com el de **FTSR** valen tots dos "0", a efectes pràctics, es igual a tenir la línia amb interrupció emmascarada.

Els noms dels bits a *stm32f4xx.h* van de **EXTI\_RTSR\_TR0** fins a **EXTI\_RTSR\_TR19** i de **EXTI\_FTSR\_TR0** fins a **EXTI\_FTSR\_TR19**.

### 9.3.6 Pending register (EXTI\_PR)

Address offset: 0x14  
Reset value: undefined

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved														PR22	PR21
									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

Bits 31:23 Reserved, must be kept at reset value.

Bits 22:0 PRx: Pending bit

0: No trigger request occurred

1: selected trigger request occurred

This bit is set when the selected edge event arrives on the external interrupt line. This bit is cleared by writing a 1 to the bit or by changing the sensitivity of the edge detector.

El registre *Pending register (PR)*, accessible com **EXTI->PR**, indica, per cadascuna de les 23 fonts d'interrupció, si s'ha generat petició d'interrupció o no.

Quan una RSI està associada a diferents línies d'interrupció, llegirem el registre **PR** per saber quina de les fonts havia demanat la interrupció. Una vegada gestionada la interrupció, hem d'esborrar la petició d'interrupció per evitar que es generi una altra interrupció tan bon punt sortim de la RSI. Per fer-ho, hem de escriure un "1" al bit associat a la línia que tenia PR a "1" i que hem atès. Si escrivim "0" a qualsevol bit del registre **PR**, el seu valor no es modificarà.

Els noms dels bits a *stm32f4xx.h* van de **EXTI\_PR\_PR0** fins a **EXTI\_PR\_PR19**.

Amb això acaba la generació de peticions d'interrupció. Aquestes peticions, que es generaran quan es doni el flanc pertinent (pujada i/o baixada) a una línia habilitada, s'envien al **NVIC** que es qui s'encarrega de fer que es cridi la RSI pertinent.

### Nested vectored interrupt controller (NVIC)

El *Nested vectored interrupt controller (NVIC)*és l'element que rep les peticions d'interrupció i, depenent de la seva prioritat i de la prioritat del procés actualment en marxa, els hi dona sortida cridant a la funció RSI associada a la interrupció.

Bàsicament, el **NVIC** té una llista de totes les fonts d'interrupció en la que s'ha de programar quina és la RSI que ha d'atendre d'interrupció i quina és la prioritat d'aquesta interrupció.

La descripció bàsica del NVIC es troba al apartat 9.1 del document "[STM32F4 Reference.pdf](#)" a partir de la pàgina 195. Dins d'aquesta descripció només es troba la taula de vectors d'interrupció que indica quin vector està associat a cada font d'interrupció. Aquest document no descriu en detall l'NVIC degut a que aquest element no és un perifèric del microcontrolador si no una part integrant del nucli processador ARM Cortex M4F. És a dir, el manual no descriu el funcionament del NVIC de la mateixa manera que no es descriuen els registres interns de la CPU.

El funcionament del'NVIC s'articula en torn a una taula que indica la funció a cridar (o més ben dit, la direcció de salt d'aquesta) per cada font d'interrupció i un conjunt de registres amb l'estat o habilitació de cada línia d'interrupció.

Els registres del'NVIC es troben definits al fitxer:

`ChibiOS_2.6.2\os\ports\common\ARMCMx\CMSIS\include\core_cm4.h`

El funcionament intern del'NVIC, igual que el funcionament intern del nucli ARM Cortex M4, excedeix els objectius de coneixements demanats per aquestes pràctiques, per tant, la seva programació la farem amb l'ajuda del sistema operatiu ChibiOS/RT.

Per programar una interrupció als registres interns NVIC farem servir la funció **nvicEnableVector**

`nvicEnableVector(Línia,CORTEX_PRIORITY_MASK(Prioritat));`

Respectar el format anterior és molt important. Fixeu-vos que la crida anterior conté una macro CORTEX\_PRIORITY\_MASK. Això es fa per que els diferents processadors ARM Cortex tenen diferent nombre de bits per escollir les prioritats. Fent servir aquesta macro, la prioritat escollida s'ajusta automàticament al nombre de bits del nostre processador.

Els paràmetres a indicar **Línia** i **Prioritat**, si volem fer servir la prioritat per defecte per la línia indicada, són, per línies GPIO gestionades per l'*External interrupt controller* (EXTI), els mostrats a la taula següent.

<b>EXTI GPIO</b>	<b>Línia</b>	<b>Prioritat</b>	<b>Vector RSI</b>
0	EXTI0_IRQHandlern	STM32_EXT_EXTI0_IRQHandler_PRIORITY	EXTI0_IRQHandler
1	EXTI1_IRQHandlern	STM32_EXT_EXTI1_IRQHandler_PRIORITY	EXTI1_IRQHandler
2	EXTI2_IRQHandlern	STM32_EXT_EXTI2_IRQHandler_PRIORITY	EXTI2_IRQHandler
3	EXTI3_IRQHandlern	STM32_EXT_EXTI3_IRQHandler_PRIORITY	EXTI3_IRQHandler
4	EXTI4_IRQHandlern	STM32_EXT_EXTI4_IRQHandler_PRIORITY	EXTI4_IRQHandler
5 a 9	EXTI9_5_IRQHandlern	STM32_EXT_EXTI5_9_IRQHandler_PRIORITY	EXTI9_5_IRQHandler
10 a 15	EXTI15_10_IRQHandlern	STM32_EXT_EXTI10_15_IRQHandler_PRIORITY	EXTI15_10_IRQHandler

Les columnes **Línia** i **Prioritat** corresponen als arguments que s'han d'introduir a la funció **nvicEnableVector** per activar una línia d'interrupció. L'última columna, **Vector RSI**, indica quin és el nom del vector associat a cada interrupció i la farem servir més tard a l'hora de definir la **Rutina de Servei d'Interrupció (RSI)** de cada interrupció habilitada.

Els valors de prioritats associats per defecte a cada línia es troben al fitxer "mcuconf.h" del vostre projecte i, en el nostre cas, es 6. Per tant, si voleu fer servir prioritats diferents de 6, es millor canviar el fitxer "mcuconf.h" que no canviar la manera de cridar la funció **nvicEnableVector**.

Observeu que les línies EXTI0 a EXTI4 tenen una línia d'interrupció per cadascuna, mentre que les línies EXTI5 a EXTI9 tenen una línia comú i, de manera similar, les línies EXTI10 a EXTI15 tenen també una línia comú. Això vol dir que no podrem tenir funcions RSI independents per línies del mateix grup comú i, per tant, quan es generi una interrupció associada a diferents línies EXTI, haurem d'examinar el registre *Pending register (PR)* per saber quina línia ha demanat la interrupció.

Si voleu saber els noms de la resta de línies d'interrupció del microcontrolador, les podeu trobar dins del fitxer *stm32f4xx.h*. Les prioritats per defecte de cada línia les trobareu, igual que en el cas de les línies EXTI, al fitxer "mcuconf.h". Els noms dels vectors d'interrupció els trobareu al fitxer "ChibiOS\_2.6.2\os\hal\platforms\STM32F4xx\hal\_lld.h".

Per deshabilitar completament una interrupció, es pot desprogramar l'**NVIC**. Per fer-ho faríem servir la funció **nvicDisableVector(Linia)** on **Linia** és la línia de la taula anterior que volem desprogramar.

## Funcions RSI

Podem observar que no li hem dit al'NVIC, en cap moment, quina es la funció que ha d'atendre la línia d'interrupció que hem programat.

Per atendre cada interrupció, l'NVIC disposa d'una taula que indica la direcció associada a la funció que ha de gestionar cada interrupció. Quan es genera una interrupció, l'**NVIC** fa una crida a la direcció de la taula corresponent a la línia que ha generat la interrupció. Si no hem definit cap funció RSI per gestionar aquest vector, es genererà una excepció.

El compilador genera automàticament el codi que omple la taula de vectors d'interrupció, per tant, no cal accedir a aquesta taula manualment. Tot i això, al compilador li cal saber quina part del nostre codi està associada a cada vector d'interrupció habilitat.

Per definir una funció associada a un vector d'interrupció farem servir la macro **CH IRQ\_HANDLER(Vector)**. Aquesta macro s'encarrega de crear un nom de funció adequat per que el compilador associï el codi que la segueix amb el nom de vector que li donem com argument. D'aquesta manera el compilador inclourà la direcció en la que es compila el codi a la posició de la taula associada al vector.

El format complet per una RSI es el següent:

```
CH_IRQ_HANDLER(Vector)
{
    CH_IRQ_PROLOGUE();
    // Start of the ISR code
    .....
    .....
    // End of the ISR code
    CH_IRQ_EPILOGUE();
}
```

L'argument **Vector**és un nombre de vector d'interrupció com pot ser qualsevol dels de la taula anterior,per exemple, EXTI0\_IRQHandler per la línia EXTI0.

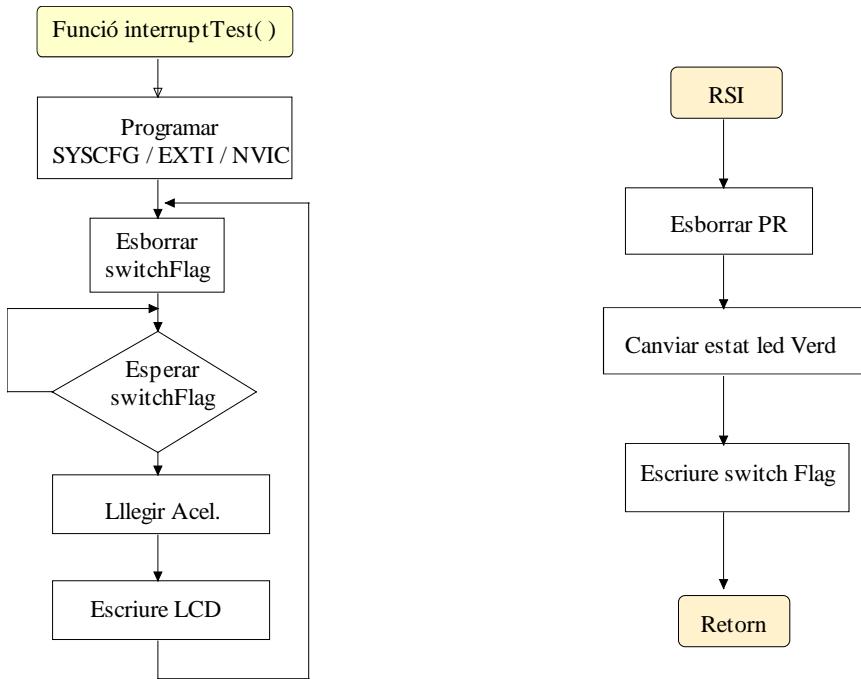
Les funcions **CH\_IRQ\_PROLOGUE( )** i **CH\_IRQ\_EPILOGUE( )** fan, respectivament, la salvaguarda i recuperació de registres del sistema ja que la RSI és asíncrona amb el funcionament de la resta del programa. El contingut de la nostra RSI, que inclou l'esborrat dels bits que calgui del *Pending register (PR)*, s'ha de trobar entre aquestes dues funcions.

## P4.2 Programa controlat per interrupcions

Per provar les interrupcions demanem generar un petit programa que, cada vegada que es polsi el polsador d'usuari:

- Canviï l'estat del **led verd**, és a dir, que s'encengui si estava apagat i s'apagui si estava encès. Podeu fer servir la operació lògica XOR "^" per fer aquesta funcionalitat.
- Actualitzi al'LCD l'estat del'eix Y del'acceleròmetre.

En general convé que les rutines de servei d'interrupció siguin tan curtes com sigui possible. Donat que tant la lectura del'acceleròmetre com l'escriptura del'LCD estan implícitament associades a períodes de temps d'espera, no convé posar aquestes funcions dins d'una RSI. És per això que es proposa fer el programa amb dues funcions amb les tasques indicades al següent diagrama de flux.



En primer lloc definirem una variable global **`switchFlag`** que comunicarà la RSI amb el programa principal. Només ens cal informació d'un bit per la funcionalitat proposada, però, en un sistema de 32 bits, la mida natural de variables enteres es de 32 bits, per tant la definirem com:

**`volatile int switchFlag;`**

Observeu el modificador "**`volatile`**". Aquest modificador és molt important quan fem servir variables que poden ser canviades dins d'una RSI. El modificador indica al compilador que aquesta variable pot canviar en qualsevol moment de manera que el compilador no pot fer cap suposició sobre l'estat actual de la variable en funció de les instruccions anteriori.

**💡** Si no féssim servir el modificador **`volatile`**, el compilador, veient que mai no s'activa `switchFlag` després d'esborrar-lo dins de la funció `interruptTest`, podria substituir tot el que ve després d'esborrar `switchFlag` per un bucle infinit el qual ocupa menys espai de codi i, segons entén el compilador, fa el mateix efecte.

La primera funció, **`interruptTest`** començarà activant i programant el *System configuration controller* (SYSCFG), l'*External interrupt controller* (EXTI) i el *Nested vectored interrupt controller* (NVIC) per tal que es cridi la RSI de la línia EXTI0 cada vegada que es doni un flanc de pujada de la línia PA0 associada al pulsador d'usuari (USR).

A continuació esborrarem `switchFlag` i posarem en marxa un bucle que només permetrà continuar el programa quan `switchFlag` canviï.

La part final de la funció farà la lectura del'acceleròmetre i l'escriptura del'LCD.

Tota la part de la funció després d'esborrar `switchFlag` haurà de trobar-se en un bucle infinit de manera que es repeteixi indefinidament tal i com es mostra al diagrama de flux.

**EP1** Escriviu una versió preliminar de la funció **void interruptTest(void)** que realitzi les operacions anteriors.

Finalment, desenvoluparem el contingut de la RSI que ha de realitzar les següents accions:

- Esborrar el bit del registre *Pending register (PR)* per indicar que s'ha atès la interrupció.
- Canviar l'estat del led Verd
- Escriure un valor diferent de zero a la variable **switchFlag**.

**EP2** Escriviu una versió preliminar del contingut de la RSI que ha de realitzar les operacions anteriors.

### Inici del treball de laboratori

En aquest punt s'inicia el treball de laboratori. Això no vol dir que no hi hagin més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

⚡ Abans que res, comproveu el bon funcionament de la placa.

Per fer aquesta pràctica disposeu dels fitxers de codi **int.c** i **int.h** continguts al fitxer **Practica-P4.zip**. Afegiu aquests fitxers al vostre directori **practica** i incorporeu-los al projecte modificant el fitxer *makefile* igual que ho vareu fer per l'LCD i l'acceleròmetre.

- 💻 Introduïu el codi demandat a EP1 i EP2 a les funcions **interruptTest** i **CH\_IRQ\_HANDLER (EXTI0\_IRQHandler)** respectivament.  
Modifiqueu la funció **main** per que en començar el programa s'inicialitzin l'LCD i l'acceleròmetre. Després, **main** ha de cridar la funció **interruptTest** de la qual no sortirà mai. Verifiqueu que el programa realitza la funció demanda.

#### **P4.3 Mesures de temporització**

Una vegada comprovat el funcionament correcte del programa, realitzarem mesures de la seva temporització. Per fer-ho, ens caldrà el cable del analitzador lògic.

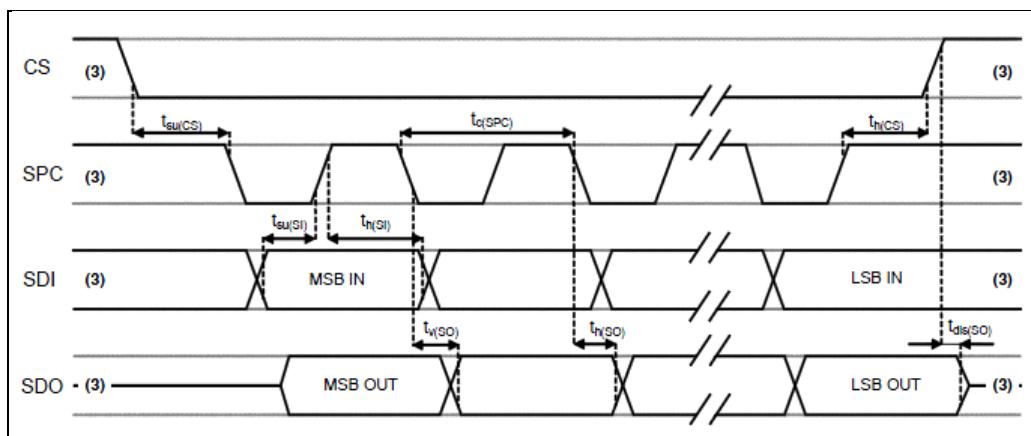
- ⚠ No oblideu connectar la línia GND del analitzador lògic a algun dels pins de referència de la placa Discovery.

La primera mesura consisteix en determinar la latència d'interrupció, és a dir, el temps que passa des de que es demana una interrupció fins que es serveix. En realitat no podem saber exactament quan s'entra a la RSI, però sí que podem saber quan canvia el led verd. Per mesurar aproximadament aquest temps, connectarem la línia del pulsador d'usuari PA0 i la línia del led Verd PD12 a dues entrades del analitzador lògic. Per agafar correctament el moment en que polsem el botó, disparem l'analitzador lògic per flanc de pujada de PA0.

- ⚡ Trobeu l'aproximació demandada a la latència d'interrupció.

A continuació farem verificacions temporals del bus SPI. Les línies GPIO associades al bus SPI (nCS, SCL, MOSI i MISO) les trobareu a la pràctica P3. Connecteu els quatre senyals al analitzador lògic sense disconnectar les línies anterior.

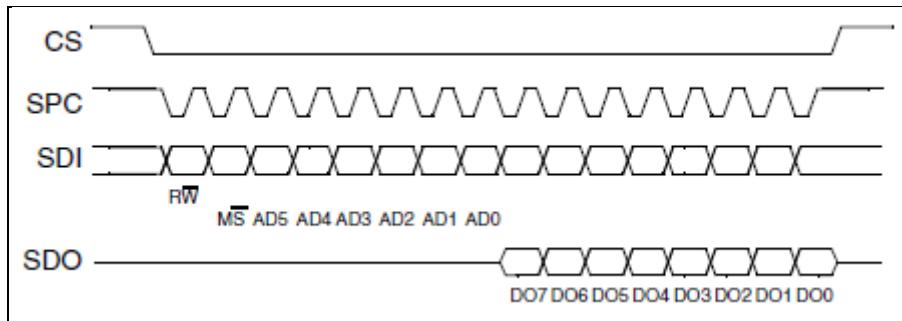
El cronograma del bus SPI en mode *slave* per l'acceleròmetre és el que mostra a la següent figura, extreta de la pàgina 12 del document "[LIS302DL.pdf](#)". El senyal SPC és el senyal de rellotge SCL. El senyal SDI és entrada del acceleròmetre, per tant MOSI (Master Out Slave Input), mentre que el senyal SDO és sortida, per tant MISO (Master Input Slave Output).



Els valors que han de complir els temps es mostra a la taula següent:

Symbol	Parameter	Value <sup>(1)</sup>		Unit
		Min.	Max.	
tc(SPC)	SPI clock cycle	100		ns
fc(SPC)	SPI clock frequency		10	MHz
tsu(CS)	CS setup time	5		ns
th(CS)	CS hold time	8		
tsu(SI)	SDI input setup time	5		
th(SI)	SDI input hold time	15		
tv(SO)	SDO valid output time		50	
th(SO)	SDO output hold time	6		
tdis(SO)	SDO output disable time		50	

Una seqüència de lectura del'acceleròmetre, com recordarem, consta d'un enviament de 8 bits amb els *flags* RW i MS i el nombre del registre a llegir (AD5..AD0), seguida de la recepció del contingut del registre DO7...DO0.



Volem mitjançant l'analitzador lògic:

- Mesurar la freqüència de rellotge del senyal SCL **fc(SPC)**
  - Verificar que la línia SDI (MOSI) apunta a una lectura del registre d'acceleració Y
  - Mesurar el temps de setup de nCS **tsu(CS)** i veure que és correcte
  - Mesurar els temps de setup **tsu(SI)** i hold **th(SI)** de SDI (MOSI) i veure que són correctes
  - Mesurar els temps **tv(SO)** i **th(SO)** de SDO (MISO) i veure que són correctes
- ⚡ Feu una captura des de la petició d'interrupció fins la lectura del acceleròmetre. Verifiqueu que es compleix tot el protocol esperat.
- ⚡ Realitzeu les mesures que es demanen i discutiu si els resultats són correctes

## ① Optimització de l'ús del temps de CPU

Si volem aprofitar al màxim el temps de CPU, l'estructura del programa es pot fer més complexa encara. Actualment el programa queda en estat d'espera mentre s'estan enviant dades tant al bus SPI com al LCD. Això es molt millorable:

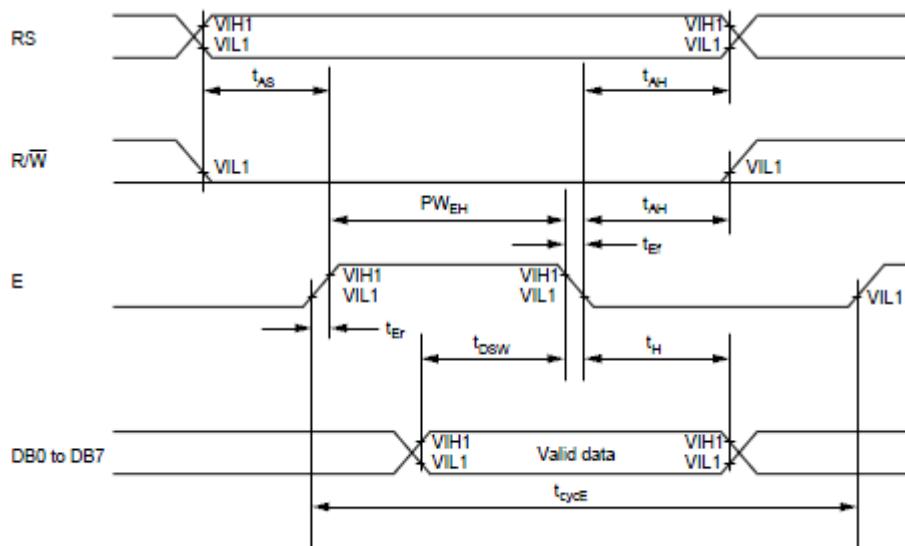
En primer lloc, el perifèric SPI es pot programar per que generi una interrupció quan acaba la transmissió actual. Per tant, el programa principal podria fer altres feines mentre dura un intercanvi mitjançant el bus SPI. De fet l'RSI del bus SPI pot encadenar els diferents accessos consecutius al bus i inicialitzar l'escriptura de l'LCD quan acaben.

En segon lloc, l'accés al LCD, tal i com l'hem programat inclou molts estats d'espera associats a l'enviament de cada *nibble*. L'enviament de comandes al LCD es podria gestionar mitjançant un sistema d'interrupcions temporitzades de manera que el programa principal pugui fer altres feines mentre dura l'escriptura sobre l'LCD. Una manera d'implementar-ho seria definir un *buffer* de bytes a enviar al LCD que ompliríem abans de cridar la funció que accedeix al LCD. Aquesta funció començaria l'enviament del primer *nibble* i posaria en marxa un temporitzador associat a una RSI per retornar després al programa principal. A partir d'aquest moment tota l'escriptura al LCD es podria fer mitjançant breus accions de la RSI que es programaria a si mateixa per tornar a ser cridada en finalitzar cada temporització fins que es buides tot el *buffer*.

No es pretén, en aquest punt, que l'estudiant realitzi aquestes millors addicionals, ni tan sols és necessari fer aquestes optimitzacions en tots els casos, inclos en programes d'aplicacions comercials. És important, però, ser conscient que tot el temps en que el processador resta a la espera d'una temporització o la finalització d'un procés es podria aprofitar, si fos necessari, per fer altres feines fent una programació adequada mitjançant interrupcions.

## P4.4 Temporització del LCD (Opcional)

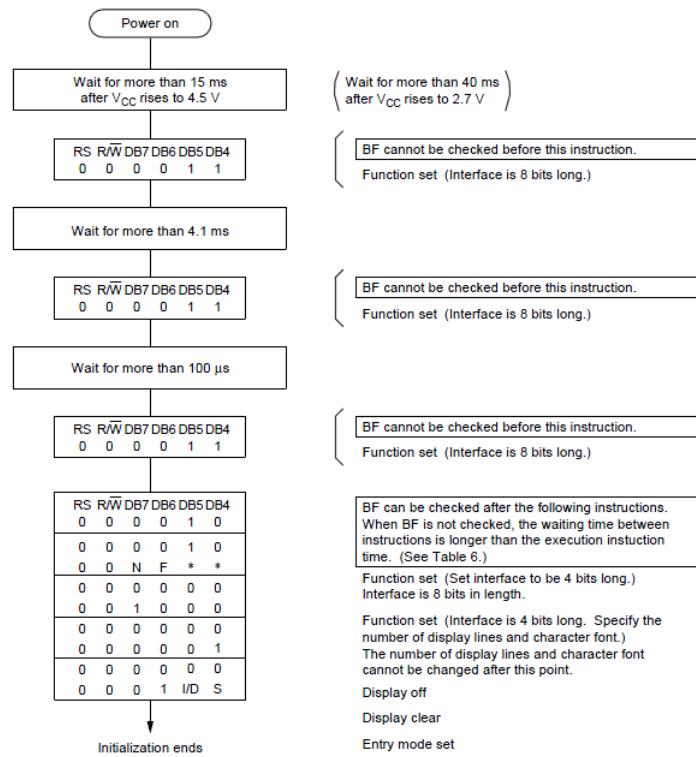
Una vegada que ens em familiaritzem amb les mesures de temporització, podem avaluar la temporització del codi del LCD que varem desenvolupar a la pràctica 2. En concret, desitgem avaluar que es compleixen els temps de setup, de hold i del controlador HD44780U del LCD  $t_{AS}$ ,  $t_{AH}$ ,  $t_{DSW}$ ,  $t_H$  i l'amplada de pols  $PWEH$ .



## Write Operation

Item	Symbol	Min	Typ	Max	Unit
Enable cycle time	$t_{cycE}$	500	—	—	ns
Enable pulse width (high level)	$PW_{EH}$	230	—	—	
Enable rise/fall time	$t_{Er}, t_{Fr}$	—	—	20	
Address set-up time (RS, R/W to E)	$t_{AS}$	40	—	—	
Address hold time	$t_{AH}$	10	—	—	
Data set-up time	$t_{DSW}$	80	—	—	
Data hold time	$t_H$	10	—	—	

També podem verificar que es compleixen els temps associats al procediment d'arrencada del LCD:



Aquí finalitza la pràctica P4.

# Pràctiques complementàries

Aquest segon bloc de pràctiques és opcional pels estudiants de **SBM**. Els grups de treball que completin la part obligatòria poden realitzar pràctiques voluntàries per pujar la nota, especialment dins del apartat d'innovació i originalitat. Les pràctiques opcionals, a diferència de les obligatòries, no cal fer-les en ordre. Per tant, els estudiants són lliures de començar per la pràctica opcional que els hi resulti de més interès.

Pels estudiants de **SEBM**, la pràctica C1 s'obligatoria i les pràctiques: C2 i C3, es poden fer voluntàriament si queda temps després de fer la resta.

Les pràctiques obligatòries demanen uns resultats i un procediment de treball molt tancat i guiat. En el cas de les pràctiques complementàries l'estructura és molt més oberta. Els estudiants poden aprofundir molt en un únic concepte associat a una pràctica o fer una petita introducció a diferents conceptes associats a diferents pràctiques complementaries.

Pels estudiants amb més inquietuds, és possible deslligar-se del marc de les pràctiques voluntàries fent un desenvolupament propi completament particular. En aquest cas, el grup de treball ha de fer una breu descripció del projecte a desenvolupar que ha de ser validat pel professor de pràctiques. D'aquesta manera es garanteix que el projecte sigui viable abans de començar-lo.

## C1 - Lectura del teclat

En aquesta pràctica es vol aprendre a fer servir el teclat com a via d'entrada pel nostre sistema basat en processador. El teclat es pot llegir per consulta i per interrupció i, en aquesta pràctica estudiarem totes dues opcions.

### Objectius de la pràctica:

- Aprendre a llegir un teclat per escanejat.
- Ser capaç de llegir el teclat mitjançant interrupcions.

### Estudis Previs:

- EP1 Versió preliminar de la funció initKeyboard
- EP2 Versió preliminar de la funció readKeyboard
- EP3 Versió preliminar de la funció intConfigKeyboard
- EP4 Versió preliminar de la RSI del teclat.

### Resultats:

- Programa de lectura del teclat per consulta
- Programa de lectura del teclat per interrupcions

### Resultats opcionals:

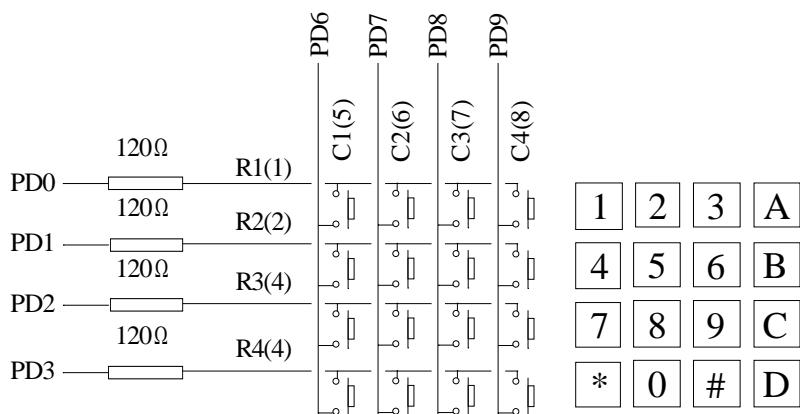
- Lectura de varies tecles simultànies
- Aplicacions Calculadora i Conversió de base

## C1.1 Descripció del teclat i procediment de lectura

Quan hem de connectar un polsador a un microcontrolador, normalment el connectem a una de les línies GPIO. Aquest és el cas del polsador d'usuari (USR) que es troba connectat a PA0.

Quan no es tracta només d'un polsador sinó d'un conjunt gran de polsadors, dedicar una línia de GPIO a cada polsador es torna poc viable degut al nombre de línies GPIO ocupades.

En el nostre cas tenim un teclat de 16 tecles. Per evitar fer ús de 16 línies de GPIO, les tecles estan distribuïdes de manera matricial com mostra la següent figura del document "[Esquemes.pdf](#)".



Les definicions de port i les línies que es fan servir es trobe definides al *board file .h* del projecte.

Com s'observa, el teclat esta distribuït en 4 fileres (R1 a R4) i 4 columnes (C1 a C4). Aquesta distribució fa que només calguin 8 línies al port D en lloc de les 16 originals. Aquesta millora és molt més important en teclats amb major nombre de tecles com és el cas d'un PC.

Quan es polsa una tecla, per exemple la tecla "6" de la filera 2 i columna 3, s'uneixen aquestes dues línies. La resta de línies romanen independents.

Les resistències de  $120\Omega$  que hi ha connectades a les fileres són una mesura de protecció en cas de que es programessin, per error, fileres i columnes a la vegada com sortides. A nivell pràctic, us podeu obrir que hi són.

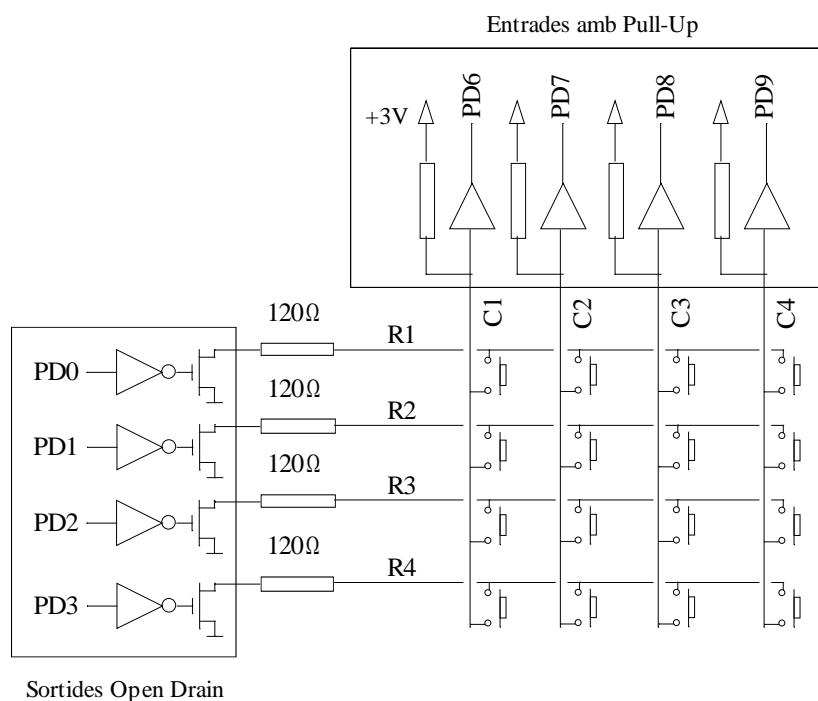
Fer servir una distribució matricial té els seus inconvenients. En primer lloc la detecció d'una tecla polsada no es instantània. És possible saber, de manera gairebé instantània, que s'ha polsat alguna tecla, però no es coneix quina s'ha polsat fins que no explorem el teclat. Això fa que calgui més temps per reconèixer les tecles que en el cas de fer servir una línia de GPIO per cada tecla. En segon lloc no és fàcil detectar qualsevol combinació de tecles a la vegada. Això ho veurem clarament quan parlem de l'exploració del teclat al proper apartat.

## Exploració del teclat

Amb 8 línies de GPIO per 16 tecles es impossible saber de manera instantània quina tecla s'ha polsat. Per conèixer la tecla polsada, s'ha d'explorar el teclat. La manera més ràpida i habitual d'explorar un teclat es configurar fileres com a sortida digital *open drain* i columnes com a entrada digital amb *Pull-Up* (o a l'inrevés).

Les entrades amb *Pull-Up* son entrades GPIO normals, però, si no connectem res i deixem l'entrada flotant, en lloc de llegir-se un valor indeterminat, llegirem el valor per defecte forçat per la resistència de *Pull-Up* que és "1".

Les sortides *Open Drain* són sortides a les que només es força un dels valors lògics, en el nostre cas el "0". Per tant, per exemple, si PD0 val "0" tindrem R1 a "0", però si PD0 val "1" tindrem R1 flotant ja que no forcem el valor "1".



A l'hora de fer l'exploració del teclat suposarem que només hi ha, com a màxim, una tecla polsada al mateix temps. Després veurem que passa si no és així.

Per explorar el teclat començarem posant PD0 a "0" i la resta de sortides "PD1 a PD3" a "1". Com que les sortides son *Open Drain*, R1 estarà a "0" i R2 a R4 estaran flotants, sense connectar a res.

Si s'ha polsat la tecla a la primera filera (R1), la seva columna respectiva quedarà a "0" ja que el "0" de sortida domina sobre el *Pull-Up*. La resta de columnes romandrà a "1" ja que només tenen el *Pull-Up*. Si la tecla s'ha polsat a un altre filera, no afecta el resultat ja que les altres fileres queden flotant. Com que suposem que no hi ha més d'una tecla polsada, només una de les columnes pot ser "0". En detectar un "0" a qualsevol columna ja podem donar per acabada l'exploració del teclat, ja que en sabem la filera (R1) i la columna. Si no s'ha polsat cap tecla a la primera filera, a totes les columnes es llegirà "1". Això vol dir que hem de continuar explorant la resta de fileres.

Repetirem el procediment explorant la segona filera, posant PD1 a "0" i la resta "PD0, PD2 i PD3" a "1". Si a cap columna es llegeix "0", continuarem amb la tercera filera i, si cal, amb la quarta.

Si acabada l'exploració de la quarta filera no s'ha llegit "0" a cap columna concloureem que no s'havia polsat cap tecla al teclat.

### ① Pulsació simultània de 2 tecles

Si només es pot polsar una tecla al mateix temps, les sortides PD0 a PD3, de fet, es podrien programar en mode *Push-Pull*. El problema es que si, encara que fos per error, polséssim dues tecles de la mateixa columna, faríem curtcircuit entre dues sortides

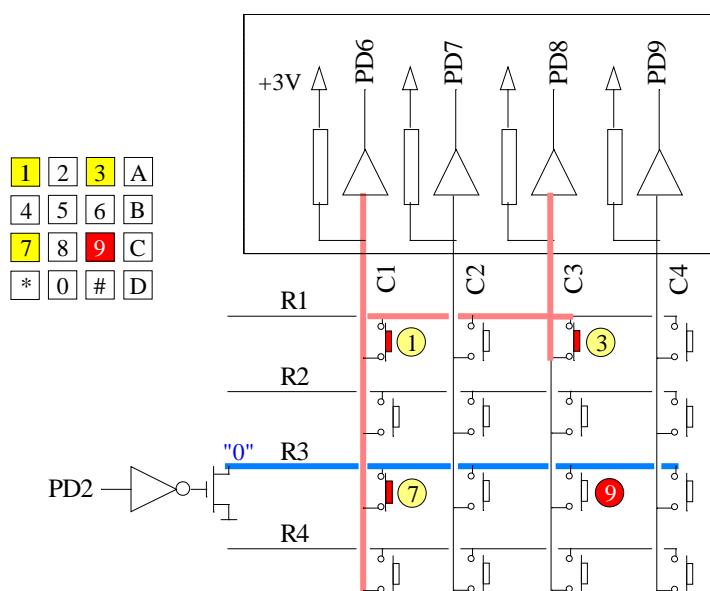
Si es polsen dues tecles, l'efecte depèn d'on estiguin situades. Si les dues tecles són a la mateixa filera, en el moment d'explorar-la, es detectaran no 1 sinó 2 "0"s. Per tant, sabrem exactament quines tecles s'han polsat. Si el sistema ha de respondre amb una única tecla, s'haurà de fixar quina columna té més prioritat.

Si les dues tecles estan a diferents fileres, degut a que l'exploració s'acaba amb la primera detecció, només es detectarà la tecla amb la filera de nombre més baix. Però, si ho desitgem, seria possible detectar les dues tecles fent sempre una exploració completa del teclat.

### ① Pulsació simultània de 3 o més tecles

El procediment d'exploració descrit permet detectar qualsevol nombre de tecles que es trobin o a la mateixa filera o a la mateixa columna. Si hi han 4 tecles polsades diferents a diferents fileres, també es poden detectar totes. Es pot detectar, simultàniament, per exemple "123A", "147\*", "159D", "246B80", fins a 6 tecles simultànies en alguns casos.

Es dona un problema, no obstant, quan es polsen 3 o més tecles i una es troba a la mateixa filera que una altra i a la mateixa columna que un'altre. Veurem, com a cas pràctic la pulsació simultània de les tecles ①, ③ i ⑦. Quan explorem la filera 1 detectarem el ① i el ③, tot correcte. Quan explorem la filera 3, el "0" de la filera 3, passarà a la columna 1, i es llegirà el ⑦, que també es correcte. No obstant, la pulsació de ① connectarà la primera columna que val "0" amb la primera filera que valdrà també "0" i, donat que el botó ③ està polsat, també llegirem "0" a la columna 3 i, erròniament llegirem la pulsació de la tecla ⑨.



Podeu experimentar amb altres casos. Per exemple, la pulsació "123A" es llegeix correctament, però la pulsació "123A9" es llegeix "123A789C".

## C1.2 Codi d'exploració del teclat

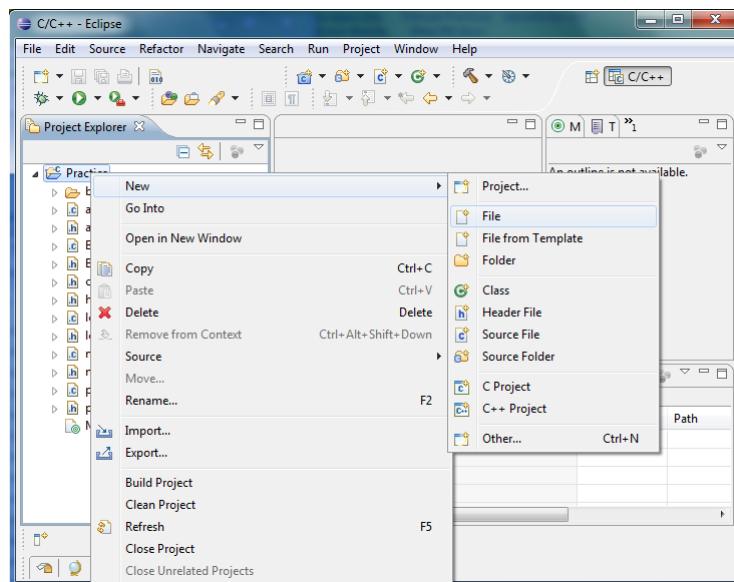
### Inici del treball de laboratori

En aquest punt s'inicia el treball de laboratori. Això no vol dir que no hi hagi més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

⚡ Abans que res, comproveu el bon funcionament de la placa.

Per desenvolupar el codi de gestió del teclat, ho podeu incorporar als fitxers que ja teniu, però seria recomanable que generessiu dos nous fitxers **keyboard.c** i **keyboard.h**. Per crear nous fitxers dins del projecte, seleccioneu el projecte Practica i obriu el menú contextual amb el botó dret del ratolí. Seleccioneu **New → File**



Genereu d'aquesta manera els dos fitxers **keyboard.c** i **keyboard.h**. Per incorporar dins del procés de compilació el fitxer **keyboard.c**, haureu de modificar el fitxer **makefile** com ja ho hem fet altres vegades a les pràctiques anteriors.

El primer que s'ha de fer és configurar el teclat, en concret, les fileres (PD0..PD3) s'han de configurar com sortida *Open Drain* i les columnes (PD6..PD9) com a entrades amb *Pull-Up*. Tota la informació sobre la programació del ports es va explicar a la pràctica P2, en concret al apartat P2.4. Recordeu que per defecte totes les línies externes es configuren en mode entrada amb Pull-Up dins la funció *boardInit()*.

EP1 Escriviu una versió preliminar de la funció que configura el teclat **initKeyboard**

La manera més fàcil de treballar amb el teclat es fer una exploració sota demanda. En concret farem una funció **readKeyboard** que retorni el codi de la tecla polsada actualment.

```
int32_t    readKeyboard(void)
```

La funció haurà de retornar un codi de tecla, si es polsa alguna tecla i un codi diferent si no se'n polsa cap. A continuació es fa una proposta de codis, però la podeu modificar si ho considereu oportú.

Tecla	Codi
1	0
2	1
3	2
A	3
CAP	32

Tecla	Codi
4	4
5	5
6	6
B	7

Tecla	Codi
7	8
8	9
9	10
C	11

Tecla	Codi
*	12
0	13
#	14
D	15

Un avantatge d'aquesta distribució de codis es que el codi retornat quan es polsa alguna tecla té exactament 4 bits (0 a 15), els 2 bits baixos corresponen a la columna mentre que els dos bits alts corresponen a la filera. El codi 32 quan no es polsa cap tecla es completament arbitrari.

El procediment recomanat per escombrar el teclat es el indicat a C1.1, és a dir, explorar filera per filera fins que alguna columna es llegeixi "0". En aquest moment es retornaria el codi de la tecla.

Si s'acaba d'explorar l'última filera sense detectar cap tecla, s'ha de retornar el codi apropiat.

Per poder fer l'exploració del teclat heu de llegir els bits d'entrada de columnes del port D. Per això haureu de fer servir el registre **Input Data Register (IDR)** del port accessible com **GPIOD->IDR**.

#### 6.4.5      GPIO port input data register (GPIOx\_IDR) (x = A..I)

Address offset: 0x10

Reset value: 0x0000 XXXX (where Xmeans undefined)

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

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 IDR[15:0]: Port input data (y = 0..15)

These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

EP2 Escriviu una versió preliminar de la funció **readKeyboard**

- ▣ Introduïu el codi de les funcions **initKeyboard** i **readKeyboard** i comproveu el seu funcionament amb un programa **main** que mostri per la pantalla LCD la tecla actual. Per fer la correspondència entre el codi de tecla i el caràcter associat podeu definir una cadena constant de 16 caràcters de manera que cada caràcter tingui com a posició el seu codi.
  
- ▣ Si teniu problemes durant l'exploració del teclat, podeu afegir un petit retard d'alguns microsegons entre el moment que es fixa una filera a "0" i el moment en que es llegeixen les columnes. Això dona temps a carregar la capacitat de les línies.
  
- ¶ Com s'ha indicat anteriorment, es poden detectar correctament entre dues i sis tecles simultànies. Un codi de tecla de 0 a 15 només permet una única tecla. Podem definir una funció **readMultikey** que retorni un codi que posi a "1" el bit corresponent a cada codi de tecla. Per exemple, la pulsació simultània de "159D" hauria de posar a "1" els bits 0,5,10 i 15 i per tant, retornar el codi 0x8421. Feu servir aquesta funció per mostrar al'LCD totes les tecles polsades simultàniament. Comproveu, finalment, els errors de detecció que poden donar-se, a vegades, quan es polsen 3 o més tecles.

### **C1.3 Gestió del teclat per interrupcions**

Per aprofundir més en la gestió del teclat podeu provar de gestionar-lo mitjançant interrupcions.

La gestió d'interrupcions a les línies GPIO va ser descrita a la pràctica P4. Podeu fer servir el cas del pulsador d'usuari com a exemple.

La funció de configuració d'interrupcions ha de fer les següents accions:

- Per detectar la pulsació d'una tecla qualsevol hauríem de posar les 4 fileres a "0" i generar interrupció per flanc de baixada de qualsevol de les columnes. Per tant, el primer és posar a "0" totes les fileres del teclat.
  
- Com que les columnes estan assignades a les línies PD6...PD9, haurem de configurar aquestes línies als senyals EXTI6...EXTI9 amb el registre **SYSCFG->EXTICR** del *System configuration controller* (SYSCFG).
  
- Amb l'*External interrupt controller* (EXTI) hem d'habilitar les interrupcions de les línies amb el registre **EXTI->IMR**, hem de seleccionar la interrupció per flanc de baixada amb el registre **EXTI->FTSR** i hem d'esborrar qualsevol interrupció pendent amb **EXTI->PR**.
  
- Les línies EXTI 5 a 9 comparteixen vector d'interrupció, per tant, només caldrà configurar un vector al *Nested vectored interrupt controller* (NVIC) per gestionar les interrupcions de les línies 6 a 9. S'haurà de fer servir la funció **nvicEnableVector** per habilitar el vector associat a la línia **EXTI9\_5\_IRQn**

**EP3** Escriviu una versió preliminar de la funció **intConfigKeyboard** que configuri el teclat per treballar per interrupcions.

Una vegada generat el codi d'inicialització, queda fer el codi de l'RSI. Podeu fer servir el codi del polysador d'usuari com a exemple, però en aquest cas farem servir el vector de les línies 5 a 9 **EXTI9\_5\_IRQHandler** en lloc del vector de la línia 0.

L'NVIC farà una crida a l'RSI quan es polsi qualsevol tecla. El codi de la RSI ha de fer, per tant, l'exploració del teclat per determinar la tecla polsada. Una vegada detectada la tecla, guardarem el seu codi a una variable global de tipus **int32\_t** amb modificador **volatile**.

- ─ La pròpia exploració del teclat que fem dins l'RSI dona lloc a flancs de baixada i pot, per tant, generar interrupcions en si mateixa. Abans d'iniciar l'exploració del teclat hem de emmasclarar les seves interrupcions amb el registre **EXTI->IMR**. Just abans de sortir de l'RSI esborrarem qualsevol interrupció pendent amb el registre **EXTI->PR** i tornarem a habilitar les interrupcions.
- ─ No oblideu que durant l'exploració només una filera pot valdre "0" i, que en finalitzar l'RSI totes les fileres s'han de posar a "0" per poder detectar la següent pulsació de qualsevol filera.

**EP4** Escriviu una versió preliminar de l'RSI del teclat.

- ─ Introduïu el codi de les funcions **intConfigKeyboard** i l'RSI del teclat. Comproveu el seu funcionament amb un programa **main** que inicialitzi el teclat i mostri periòdicament per la pantalla LCD el contingut de la variable global que actualitza l'RSI.

#### C1.4 Més propostes optatives

Com ja s'ha comentat, els apartats optatius són molt oberts. Aquí es donen algunes propostes de programes que es poden fer amb els coneixements adquirits.

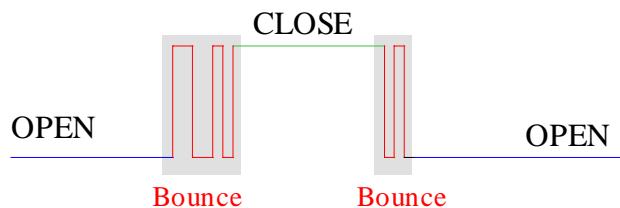
**Calculadora:** Com que ja sabem fer servir tant el teclat com l'LCD, és possible desenvolupar una calculadora amb les quatre operacions bàsiques. Per simplicitat podeu fer que treballi només amb nombres sencers. També es pot treballar en coma flotant, però, en aquest cas, haureu de desenvolupar una funció equivalent a **itoa** per a tipus **float** o **double**.

**Conversió de base:** Aquesta proposta és semblant a l'anterior. Es proposa fer un programa que permeti introduir un nombre en base **hexadecimal**, **decimal**, **octal** o **binària** i permeti convertir a una base diferent de la original. Com que l'LCD es de 16 díigits d'amplada, ens podem restringir a nombres de 16 bits. Per simplicitat podeu fer servir només nombres sense signe.

### ⓘ Button Bounce

Quan treballieu amb el teclat o l'interruptor d'usuari, és important tenir en compte que es pot donar un fenomen conegut com *Button Bounce* (rebot del interruptor).

Quan polem un interruptor o una tecla, es poden fer diferents contactes d'obertura i tancament fins que s'estabilitza l'estat del interruptor.



En el cas del teclat, això vol dir que, després de processar una tecla, és bona idea esperar un temps abans de processar una altra tecla per tal de donar temps a que s'hagin acabat els rebots. Amb uns 10 µs a 100 µs de temps acostuma a haver-n'hi prou aquest valor, però no es exacte.

## C2 - Convertidor A/D

En aquesta pràctica es vol aprendre a fer servir un Convertidor Analògic a Digital (ADC). De fet, el STM32F407 té 3 convertidors A/D de 12 bits. En aquesta pràctica farem servir només el primer d'ells, l'ADC1, per convertir el valor analògic del potenciómetre (de color groc a la placa de pràctiques) que es troba connectat a l'entrada analògica IN8 associada a la línia GPIO PB0.

### Objectius de la pràctica:

- Aprendre a llegir valors analògics amb el convertidor A/D.

### Estudis Previs:

- EP1 Programació del rellotge del ADC a C2.2
- EP2 Programació del temps de lectura (*Sample Time*) a C2.2
- EP3 Versió preliminar d'**initADC** a C2.2
- EP4 Versió preliminar de **readIN8** o **readChannel** a C2.3
- EP5 Lectura de la temperatura del integrat a C2.5
- EP6 Versió preliminar de **readVdd** a C2.6

### Resultats:

- Programa de lectura de la tensió al potenciómetre
- Programa de lectura de la temperatura del microcontrolador
- Programa de lectura de la tensió d'alimentació

### Resultats opcionals:

- Fer servir la funció **readChannel**
- Llegir la temperatura considerant el calibrat de Vdd a C2.6

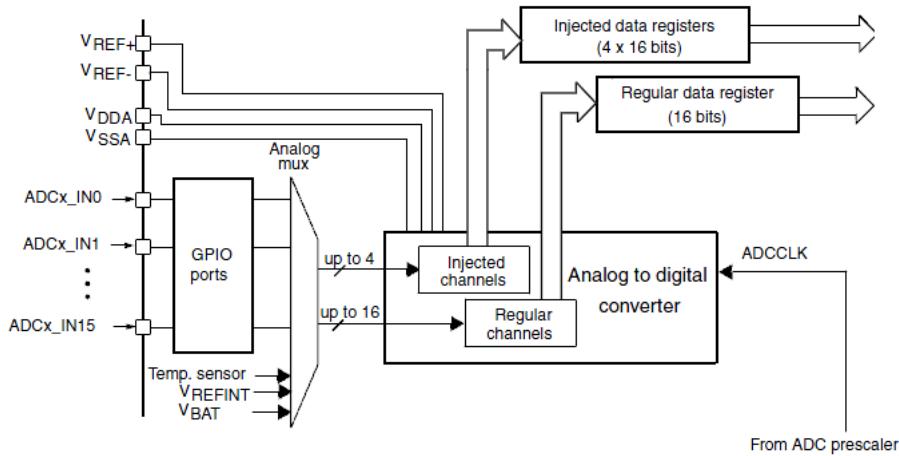
### C2.1 Descripció del convertidor ADC

La descripció completa dels convertidors ADC es troba a partir de la **pàgina 207** del document "[STM32F4 Reference.pdf](#)". Els ADC que incorpora el microcontrolador que fem servir a les pràctiques te moltes capacitats. Entre elles:

- Resolució configurable de 12, 10, 8 o 6 bits
- Conversió de dos seqüències de canals: regulars i injectats
- Conversió contínua o única
- Sincronització de la conversió amb diferents fonts
- Sincronització entre els tres ADC per fer conversions al mateix temps
- Transferència de dades de conversió mitjançant DMA
- Generació de diferents tipus d'interrupcions

Tenir un sistema molt potent té l'inconvenient que la seva programació és complexa fins i tot per fer feines senzilles com la lectura d'un únic canal d'entrada. En els següents apartats s'intentarà explicar el mínim necessari per poder fer conversions simples d'un canal amb l'ADC 1.

L'estructura bàsica d'un ADC del microcontrolador es mostra a la següent figura:



Les alimentacions coincideixen, en el nostre cas, amb les referències, per tant  $V_{REF+} = V_{DDA} = 3V$  i  $V_{REF-} = V_{SSA} = 0V$ .

Disposem de 19 entrades pel nostre ADC, 16 estan associades a pins de GPIO que poden treballar com entrades analògiques IN0...IN15 i 3 corresponen a un sensor de mesura de temperatura (IN16), una tensió de referència (IN17) i una mesura de la tensió de la pila de *backup* que pot portar el sistema (IN18). Les últimes 3 entrades IN16, IN17 i IN18 només son disponibles, però, al convertidor ADC 1. Això no ens afecta per que és l'únic que farem servir.

Per fer conversions podem fer dues seqüències de canals, una **regular** amb un màxim de 16 canals i una de canals **injectats** amb un màxim de 4 canals. Els canals es poden repetir varíes vegades dins d'una llista; així, la llista de canals regulars pot ser, per exemple:

IN1	IN0	IN1	IN4	IN17	IN0
-----	-----	-----	-----	------	-----

Això vol dir que primer es convertirà el canal 1, després el 0, després tornem al '1, i seguim fins que acabem al final amb el canal 0.

Els resultats de les conversions dels canals **regulars** s'emmagatzemen tots al registre *Regular Data Register* (DR), per tant, si es fa una conversió d'una seqüència de canals, s'ha de llegir cada conversió abans de que s'escrigui la següent. Per fer-ho correctament l'ADC incorpora *flags* que s'activen en acabar la conversió i es pot, també, generar interrupcions al final de cada conversió o enviar els resultats a la memòria RAM mitjançant DMA.

En el cas dels canals **injectats**, els resultats del màxim de 4 canals de la llista s'emmagatzemen en 4 registres independents denominats *Injected Data Registers* (JDR1..JDR4). Per tant, es pot esperar la conversió de tota la seqüència abans de llegir les dades.

L'ADC té el seu propi rellotge que s'obté mitjançant un divisor, del rellotge del bus perifèric APB2 que es el mateix del perifèric SPI 1 que fa servir l'acceleròmetre.

## C2.2 Configuració del ADC 1

En aquest apartat ens concentrarem en la manera de configurar el convertidor ADC 1 per fer conversions d'un únic canal regular. En concret, del canal IN8 associat al potenciómetre de la placa.

❶ En primer lloc s'ha d'activar, com ja es costum, el rellotge del perifèric. El convertidor ADC 1 penja del bus APB2 que és el mateix del que penja el perifèric SPI 1 amb el que ja hem treballat a la pràctica P3. Hem de treballar, per tant, amb el registre RCC->APB2ENR, descrit a **P3.2**, però en aquest cas el bit a activar és **ADC1EN**.

❷ En segon lloc hem de fixar la freqüència de rellotge del convertidor. L'ADC penja del bus APB2 que opera a 84MHz i que es divideix amb el *prescaler* del ADC. El control del *prescaler* es troba al registre *Common Control Register* (CCR). Donat que 's un registre comú als tres ADCs, penja de l'estruatura comú ADC i s'hi accedeix com ADC->CCR.

### 10.13.16 ADC common control register (ADC\_CCR)

Address offset: 0x04 (this offset address is relative to ADC1 base address + 0x300)

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBALE	Reserved				ADCPRE	
								rw	rw					rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA[1:0]	DDS	Res.	DELAY[3:0]				Reserved				MULTI[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

Bits 17:16 ADCPRE: ADC prescaler

Set and cleared by software to select the frequency of the clock to the ADC. The clock is common for all the ADCs.

Note: 00: PCLK2 divided by 2  
01: PCLK2 divided by 4  
10: PCLK2 divided by 6  
11: PCLK2 divided by 8

Els bits de ADCPRE es referencien amb ADC\_CCR\_ADCPRE\_0 i ADC\_CCR\_ADCPRE\_1 per cada bit i amb ADC\_CCR\_ADCPRE per tots dos a la vegada *stm32f4xx.h*.

Les freqüències màximes i mínimes de rellotge per l'ADC es troben a la pàgina 118 del document "[STM32F407 Datasheet.pdf](#)". Si no teniu una altra preferència, podeu escollir el valor del *prescaler* PCLK2 que més aproximi la freqüència del ADC al valor típic per una alimentació de 3V.

**EP1** Escolliu un valor per PCLK2 i determineu el valor a programar als bits 16 i 17 del registre *Common Control Register* (CCR).

❸ En tercer lloc s'ha de determinar el temps de lectura  $T_S$  (*Sample Time*). El temps de lecturaés el temps que l'ADC roman connectat al canal d'entrada per estabilitzar el seu valor abans d'iniciar-se la seva conversió. El control és independent per cada canal i es programa mitjançant dos registres *ADC Sample Time Registers 1,2* (SMPR1 i SMPR2).

#### 10.13.4 ADC sample time register 1 (ADC\_SMPR1)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				SMP18[2:0]			SMP17[2:0]			SMP16[2:0]			SMP15[2:1]		
				rw	rw	rw									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

#### 10.13.5 ADC sample time register 2 (ADC\_SMPR2)

Address offset: 0x10

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				SMP9[2:0]			SMP8[2:0]			SMP7[2:0]			SMP6[2:0]		
				rw	rw	rw									
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SMP5_0	SMP4[2:0]			SMP3[2:0]			SMP2[2:0]			SMP1[2:0]			SMP0[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

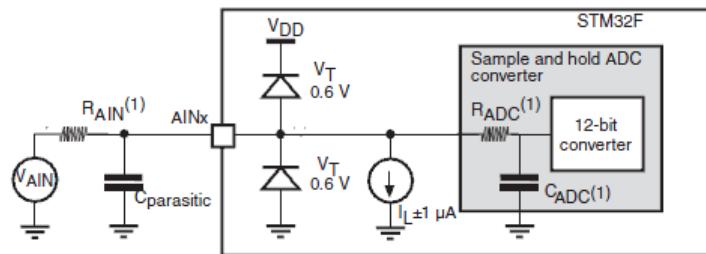
##### SMPx[2:0]: Channel x sampling time selection

These bits are written by software to select the sampling time individually for each channel.  
During sample cycles, the channel selection bits must remain unchanged.

- Note:* 000: 3 cycles
- 001: 15 cycles
- 010: 28 cycles
- 011: 56 cycles
- 100: 84 cycles
- 101: 112 cycles
- 110: 144 cycles
- 111: 480 cycles

El bits del registre ADC1->SMPR1 a *stm32f4xx.h* tenen noms com ADC\_SMPR1\_SMP10 o ADC\_SMPR1\_SMP10\_0, i els de ADC1->SMPR1 tenen noms com ADC\_SMPR2\_SMP2 o ADC\_SMPR2\_SMP2\_1 .

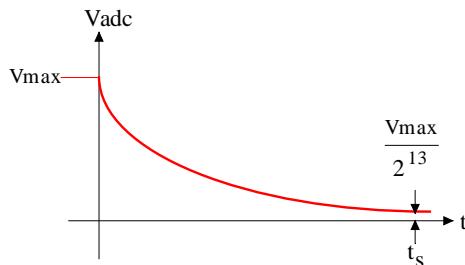
L'entrada IN8 està associada al temps de lectura  $t_S$  fixat per SMP8 que es troba al registre SMPR2 accessible com ADC1->SMPR2. El valor que escollim serà un compromís entre el temps de conversió i l'exactitud del resultat. Per estudiar el temps més adequat per la lectura del potenciòmetre podem fer servir el model de connexió del'ADC que es troba a la pàgina 121 del document "[STM32F407 Datasheet.pdf](#)".



La resistència  $R_{ADC}$  la trobem a la mateixa taula que es trobava la freqüència d'operació del'ADC i és, com a màxim,  $6k\Omega$ . La capacitat  $C_{ADC}$  es troba a la mateixa taula i és, típicament,  $4pF$ . La resistència  $R_{AIN}$  depèn de la posició del potenciómetre i de la resistència sèrie de  $120\Omega$  que té el seu cursor tal i com es veu al document "[Esquemes.pdf](#)". Per simplicitat, considerarem una capacitat paràsita petita comparada amb  $C_{ADC}$ .

En el moment d'iniciar-se el temps de lectura, es donarà una càrrega RC exponencial del condensador  $C_{ADC}$  a través de les resistències en sèrie  $R_{AIN}$  i  $R_{ADC}$ . Com que la nostra conversió es a 12 bits, volem que al final del temps de lectura el valor es trobi a menys de mig bit ( $\text{Rang}/2^{13}$ ) del valor final. Considerarem el cas pitjor en el que el condensador fa una transició de  $V_{DD}$  a 0 per calcular el temps necessari per tenir 12 bits de precisió.

Escollirem el primer valor de nombre de cicles de lectura que es trobi per sobre del temps obtingut.



El temps total de conversió depèn del temps de lectura ( $t_s$ ) i del temps que triga en fer-se la conversió en sí mateixa ( $t_c$ ) que es fa per aproximacions successives i, per tant, triga un cicle de rellotge per bit. Per tant, la fórmula del temps total es:

$$T_{Tot} = t_s + t_c = T(n_s + n_B)$$

On  $T$  es el període del rellotge del'ADC,  $n_s$ és el nombre de cicles de rellotge del temps de lectura (Sample) i  $n_B$ és el nombre de bits de conversió que per defecte és 12 però que pot ser també 10, 8 o 6.

#### EP2 Determineu:

- La constant de temps  $\tau$  del circuit de càrrega de l'entrada del'ADC
- El temps de lectura  $T_s$  (Sample) per tenir exactitud de 12 bits
- El nombre de cicles de lectura a codificar en SMP8 i el seu codi binari de 3 dígitos associat
- El temps total de conversió

Si volguéssim llegir més canals, a part del canal del potenciòmetre IN8, hauríem de configurar de manera similar el temps de lectura de cadascun d'ells.

④ En quart lloc hem de configurar la línia GPIO PB0 per que operi com a entrada analògica.

A la **pàgina 47** del document "[STM32F407 Datasheet.pdf](#)" trobem les funcions associades a la línia GPIO PB0. Com podem veure, la seva funció addicional es ADC12\_IN8. Es a dir, entrada 8 del convertidor de 12 bits.

Table 6. STM32F40x pin and ball definitions (continued)

Pin number					Pin name (function after reset) <sup>(1)</sup>	Pin type	I/O structure	Notes	Alternate functions		Additional functions	
LQFP64	LQFP100	LQFP144	UFBGA176	LQFP176								
23	32	43	R3	53	PA7	I/O	FT	(4)	SPI1_MOSI/TIM8_CH1N/TIM14_CH1/TIM3_CH2/ETH_MII_RX_DV/TIM1_CH1N/RMII_CRS_DV/EVENTOUT		ADC12_IN7	
24	33	44	N5	54	PC4	I/O	FT	(4)	ETH_RMII_RX_D0/ETH_MII_RX_D0/EVENTOUT		ADC12_IN14	
25	34	45	P5	55	PC5	I/O	FT	(4)	ETH_RMII_RX_D1/ETH_MII_RX_D1/EVENTOUT		ADC12_IN15	
26	35	46	R5	56	PB0	I/O	FT	(4)	TIM3_CH3/TIM8_CH2N/OTG_HS_ULPI_D1/ETH_MII_RXD2/TIM1_CH2N/EVENTOUT		ADC12_IN8	

Per que PB0 operi com a entrada analògica, hem de configurar el registre MODER del port BGPIOB->MODER per tal que la seva línia 0 es configuri com a analògica. Aquest registre el podeu trobar descrit a la practica 2, apartat P2.4. Per si de cas, desconecteu el seu *Pull-Up*.

⑤ En cinquè i últim lloc, hem d'activar l'ADC. Per fer-ho, només cal posar a "1" el bit 0 (ADON) del registre *ADC Control Register 2* (CR2) accessible com ADC1->CR2.

#### 10.13.3 ADC control register 2 (ADC\_CR2)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserve d	SWST ART		EXTEN		EXTSEL[3:0]				reserve d	JSWST ART		JEXTEN		JEXTSEL[3:0]	
	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
reserved				ALIGN		EOCS	DDS	DMA	Reserved				CONT	ADON	
				rw	rw	rw	rw	rw					rw	rw	

Bit 0 ADON: A/D Converter ON / OFF

This bit is set and cleared by software.

Note: 0: Disable ADC conversion and go to power down mode  
1: Enable ADC

EP3 Escriviu una versió preliminar d'una funció **initADC** per inicialitzar l'ADC 1

## C2.3Lectura d'un canal amb el ADC 1

El procediment del'apartat anterior explica com programar l'ADC abans de fer cap conversió. En aquest apartat explicarem com realitzar la lectura d'un dels canals. Recordeu que el temps de lectura t<sub>s</sub> d'aquest canal ha d'haver estar programat prèviament.

Com ja s'ha indicat, l'ADC està pensat per realitzar seqüències de lectures de dos tipus, **regulars** (fins a 16 lectures) i **injectades** (fins a 4 lectures). Fer una única lectura correspon a tenir una seqüència regular de només una lectura.

La seqüència regular de conversions es programa amb els registres *ADC Regular Sequence Registers* 1, 2, 3 (SQR1,SQR2 i SQR3)

10.13.9 ADC regular sequence register 1 (ADC_SQR1)																													
Address offset: 0x2C														Reset value: 0x0000 0000															
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16																													
Reserved																													
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0																													
SQ16_0								SQ15[4:0]				SQ14[4:0]				SQ13[4:0]													
rw		rw		rw		rw		rw		rw		rw		rw															
Bits 23:20 L[3:0]: Regular channel sequence length																													
These bits are written by software to define the total number of conversions in the regular channel conversion sequence.																													
0000: 1 conversion																													
0001: 2 conversions																													
...																													
1111: 16 conversions																													
10.13.10 ADC regular sequence register 2 (ADC_SQR2)																													
Address offset: 0x30														Reset value: 0x0000 0000															
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16																													
Reserved		SQ12[4:0]				SQ11[4:0]				SQ10[4:1]																			
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0		rw		rw		rw		rw		rw		rw		rw															
SQ10_0								SQ9[4:0]				SQ8[4:0]				SQ7[4:0]													
rw		rw		rw		rw		rw		rw		rw		rw															
10.13.11 ADC regular sequence register 3 (ADC_SQR3)																													
Address offset: 0x34														Reset value: 0x0000 0000															
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16																													
Reserved		SQ6[4:0]				SQ5[4:0]				SQ4[4:1]																			
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0		rw		rw		rw		rw		rw		rw		rw															
SQ4_0								SQ3[4:0]				SQ2[4:0]				SQ1[4:0]													
rw		rw		rw		rw		rw		rw		rw		rw															

Podeu deduir els noms dels bits d'aquest registres o fer una ullada al document *stm32f4xx.h*.

El primer registre SQR1 accessible com ADC1->SQR1 conté el camp *Regular Channel Sequence Length* L[3:0] de 4 bits on s'indica la longitud de la seqüència regular. Com que només volem fer una conversió, deixem el valor per defecte de 0000.

La resta de camps del tres registres son de tipus SQxx on xx és el nombre d'ordre dins de la seqüència. Com que la nostra seqüència es de longitud 1, només cal omplir el camp SQ1 que es troba al registre SQR3 accessible com ADC1->SQR3. En aquest camp posarem el nombre del canal a convertir que, pel cas del potenciòmetre es el 8.

Una vegada fixada la seqüència de conversió, el següent és iniciar la conversió. Per fer-ho, hem d'activar el bit **SWSTART** del registre *ADC Control Register 2* (CR2). Aquest registre ja l'hem fet servir abans per activar el ADC amb el bit **ADON**.

Bit 30 <b>SWSTART:</b> Start conversion of regular channels
This bit is set by software to start conversion and cleared by hardware as soon as the conversion starts.
0: Reset state
1: Starts conversion of regular channels
<i>Note: This bit can be set only when ADON = 1 otherwise no conversion is launched.</i>

Com es veu, aquest bit el posem a "1" per iniciar la conversió. El bit es posa automàticament a "0" tot just començar el procés de conversió.

La manera més senzilla, encara que no la més eficient, de llegir el resultat de la conversió és esperar a que l'ADC acabi. Per saber quan acaba la conversió es pot llegir el *ADC Status Register* (SR) accessible com ADC1->SR.

#### 10.13.1 ADC status register (ADC\_SR)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								OVR	STRT	JSTRT	JEOC	EOC	AWD		
								rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0		

Bit 1 **EOC:** Regular channel end of conversion

This bit is set by hardware at the end of the conversion of a regular group of channels. It is cleared by software or by reading the ADC\_DR register.

0: Conversion not complete (EOCS=0), or sequence of conversions not complete (EOCS=1)  
1: Conversion complete (EOCS=0), or sequence of conversions complete (EOCS=1)

Per tant, podem romandre en un bucle d'espera fins que el bit EOC passi a valdre "1".

Una vegada feta la conversió, com ja s'ha comentat anteriorment, el resultat de la conversió, amb 12 bits de precisió, restarà al registre *Regular Data Register* (DR) que es accessible com ADC1->DR.

EP4 Escriviu una versió preliminar d'una funció <b>readIN8</b> que retorni el valor llegit del canal IN8 associat al potenciòmetre.
---

- 💡 Si voleu ser més genèrics podeu fer una funció **readChannel** que prengui com argument el nombre de canal a llegir, d'aquesta manera no caldrà fer una funció nova per llegir cada canal del ADC.

## C2.4Prova del codi

### Inici del treball de laboratori

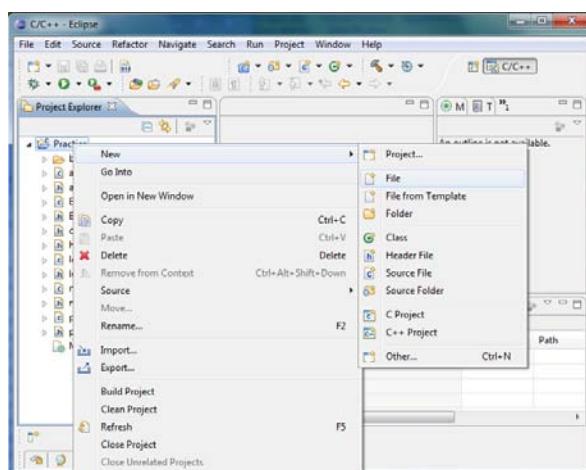
En aquest punt s'inicia el treball de laboratori. Això no vol dir que no n'hi hagin més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

- ⚡ Abans que res, comproveu el bon funcionament de la placa.

Per desenvolupar el codi de gestió del'ADC, ho podeu incorporar als fitxers que ja teniu, però seria recomanable que generessiu dos nous fitxers **analog.c** i **analog.h**. No es recomana fer servir el nom **adc** per que podria entrar conflicte amb altres fitxers de ChibiOS.

Per crear nous fitxers dins el projecte, seleccioneu el projecte Practica i obriu el menú contextual amb el botó dret del ratolí. Seleccioneu **New → File**



Genereu d'aquesta manera els dos fitxers **analog.c** i **analog.h**. Per incorporar dins el procés de compilació el fitxer **analog.c**, haureu de modificar el fitxer **makefile** com ja ho hem fet altres vegades a les pràctiques anterior.

- 💻 Introduïu el codi de les funcions **initADC** i **readIN8** (o **readChannel** ).  
Feu un programa que mostri per pantalla de manera periòdica la lectura del canal 8 del'ADC.  
Observeu com canvia entre 0 i 4095 en girar el potenciómetre.

## **C2.5Lectura de la temperatura del microcontrolador**

Tal i com s'ha descrit al apartat C2.1, un dels canals disponibles al convertidor ADC 1 és la lectura de la temperatura del microcontrolador (Canal IN16).

La descripció bàsica del sensor de temperatura es troba a la pàgina 229 del document "[STM32F4 Reference.pdf](#)". La resposta del sensor de temperatura és una tensió que es considera lineal entre -40°C i 125°C. La recta es pot aproximar, per tant per dos valors. Aquests són el seu pendent (*Avg\_Slope*) i el valor de pas per 25°C (*V25*). Per tant, a partir de la sortida del sensor *Vsense*, la temperatura del xip serà:

$$T(^{\circ}\text{C}) = 25^{\circ}\text{C} + \frac{V_{\text{Sense}} - V25}{\text{Avg\_Slope}}$$

Les característiques del sensor de temperatura, i en concret els valors particulars de *V25* i de *Avg\_Slope* es troben a la **pàgina 123** del document "[STM32F407 Datasheet.pdf](#)".

Al mateix document s'indica que haurem tenir un temps de lectura mínim (Sample Time) de 10μs per llegir el sensor de temperatura.

Per poder llegir la temperatura amb l'ADC haurem de modificar la funció **initADC** de manera que es facin les següents accions:

- Ajustar el temps de lectura del canal IN16 que correspon al sensor de temperatura per que aquest sigui de 10μs o més. Per fer-ho haurem de modificar el registre ADC1 -> SMPR1.
- Activar el sensor de temperatura mitjançant el bit TSVREFE del *Common Control Register* (CCR) accessible com ADC -> CCR.

<p>Bit 23 <b>TSVREFE</b>: Temperature sensor and <math>V_{\text{REFINT}}</math> enable This bit is set and cleared by software to enable/disable the temperature sensor and the <math>V_{\text{REFINT}}</math> channel. 0: Temperature sensor and <math>V_{\text{REFINT}}</math> channel disabled 1: Temperature sensor and <math>V_{\text{REFINT}}</math> channel enabled</p>
--

Fixeu-vos que **TSVREFE** també activa la referència de tensió, per tant, aquest bit també s'ha d'activar per l'apartat C2.6.

Una vegada modificat **initADC**, haurem de crear una funció **readT** que retorni la temperatura actual llegint el canal 16 de la mateixa manera que s'ha llegit el canal IN8 del potenciómetre. Si fem servir nombres sencers podem retornar la temperatura en dècimes de grau de manera que, per exemple 205 correspongui a 20,5°C.

Tot i que podríem fer servir variables amb coma flotant, es important saber treballar en coma fixa. Per tant, totes les operacions per trobar la temperatura les heu de fer fent servir variables senceres i no de coma flotant. Aixó vol dir que heu de tenir cura en la propagació de errors numèrics.

**EP5** Escriviu una versió preliminar de les modificacions de **initADC** i de la funció **readT** per tal de poder llegir la temperatura del xip amb un decimal de precisió.

- Introduïu el codi de la funció **readT** i les modificacions d'**initADC** i **readIN8**.

Feu un programa que mostri per pantalla de manera periòdica la temperatura actual del xip amb un decimal de precisió.

## C2.6Calibrat del'ADC

Com s'ha indicat anteriorment, l'ADC converteix en valors digitals de 12 bits (0 a 4095) els valors de tensió entre GND i Vdd.

Sabem que Vdd val uns 3V, però, si volem fer conversions precises, hem de saber quin es exactament el seu valor.

El convertidor ADC té el canal IN17 connectat a una referència de tensió. El valor d'aquesta referència i el temps mínim de mesura (*Sample Time*) es troba a la **pàgina 123** del document "[STM32F407 Datasheet.pdf](#)".

Llegint el valor del canal de referència IN17, donat que sabem a quina tensió correspon, podem saber la relació entre el valor de sortida digital del convertidor i el valor de tensió d'entrada.

Amb aquesta relació podem saber la tensió d'alimentació que es la que correspon a la lectura màxima 4095. Amb aquesta informació podem calibrar els resultats llegits per tots els canals del ADC i obtenir resultats exactes o, al menys, tan exactes com és la referència que hem fet servir.

**EP6** Escriviu una versió preliminar d'una funció **readVdd** que retorni el valor de Vdd en milivolts.  
No feu servir coma flotant en els càlculs.

- Introduïu el codi de la funció **readVdd** i feu un programa que mostri en pantalla la tensió d'alimentació. No oblideu de modificar la funció **initADC** per programar el temps de lectura del canal 17.

- ¶ Podeu millorar ara la lectura de la temperatura del integrat fent servir el calibrat de la tensió d'alimentació per obtenir resultats més exactes.

## C3 - Encoder

En aquesta pràctica es vol aprendre a fer servir un *encoder* de quadratura per donar un valor d'entrada al sistema. La gestió es farà mitjançant rutines de servei d'interrupció RSI similars a la que es proposa a la pràctica P4.

### Objectius de la pràctica:

- Aprendre a llegir l'*encoder* de quadratura.

### Estudis Previs:

- EP1 Versió preliminar de la funció initEncoder
- EP2 Versió preliminar del codi de les dues RSI que cal fer servir

### Resultats:

- Programa que llegeix l'*encoder*

### Resultats opcionals:

- Modificació de les RSI per treballar amb 4 flancs

### C3.1 Descripció del'*encoder* de quadratura

Un *encoder* de quadratura es un dispositiu rotatiu que permet enregistrar moviments de rotació relatius. L'*encoder* dona una resposta en forma de polsos quan gira de manera que podem determinar l'angle que ha girat contant els polsos.

A la nostra placa, l'*encoder* es troba sota el potenciómetre de la pràctica V2, a la esquerra del teclat. Es troba identificat amb una marca blava al seu element de control. Podeu comprovar, fent-lo girar, que te dues diferències importants amb els dos potenciómetres que te a sobre:

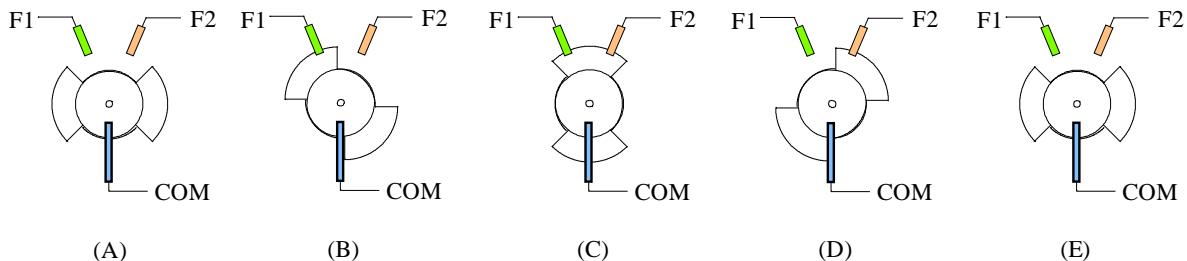
En primer lloc, te rotació contínua, es a dir, no es bloqueja en cap moment i, per tant, podem donar qualsevol nombre de voltes en qualsevol direcció. En segon lloc, te punts discrets de parada, en concret 24. Només es pot parar en aquests punts, no en cap posició intermèdia.

Els *encoders*, manuals, com el de la placa son molt útils per donar ordres d'entrada quantitatives en sistemes digitals com nivell de so, greus o baixos en un sistema d'àudio o velocitat en un sistema de control de motors. Com a mètode d'entrada superen als potenciómetres en molts aspectes:

- Donen directament valors digitals sense que calgui un convertidor A/D
- Son intrínsecament relatius, per tant, es pot prendre qualsevol posició com a zero i, per tant, es pot fer servir un únic *encoder* per ajustar diferents valors sempre que no sigui al mateix temps.
- Poden tenir qualsevol precisió només augmentant el nombre de voltes
- Tenen menys desgast que els potenciómetres

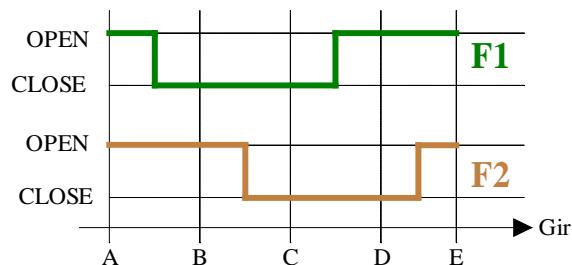
Altres tipus d'*encoders* es fan servir per mesurar la rotació de motors o altres elements rotatius. En concret, en els ratolins d'ordinador de bola, ara substituïts pels led o laser, el moviment en els eixos X i Y es mesurava amb dos *encoders* de quadratura.

A nivell de construcció hi ha diverses maneres de fabricar un *encoder*. A la figura següent es mostra un exemple senzill d'un *encoder* que té 2 polsos per volta. Aquest encoder està basat en un disc rotatiu metàl·lic retallat permanentment connectat a un terminal comú (COM). Sobre aquest disc tenim dos contactes que anomenarem Fase 1 (F1) i Fase 2 (F2).



En començar, el disc no toca cap contacte i tenim els circuits F1 i F2 oberts. Quan comença a girar cap a la dreta, arriba un punt en què F1 queda connectat al disc, més endavant es connecta també F2. A continuació es disconnecta F1 i, finalment es disconnecta F2 tornant a un estat equivalent al inicial. No es el estat inicial per que, com podeu observar del dibuix, només hem fet mitja volta. Es per això que aquest *encoder* es de dos polsos per volta.

La següent figura mostra l'estat de les dues fases en funció del gir durant mitja volta.

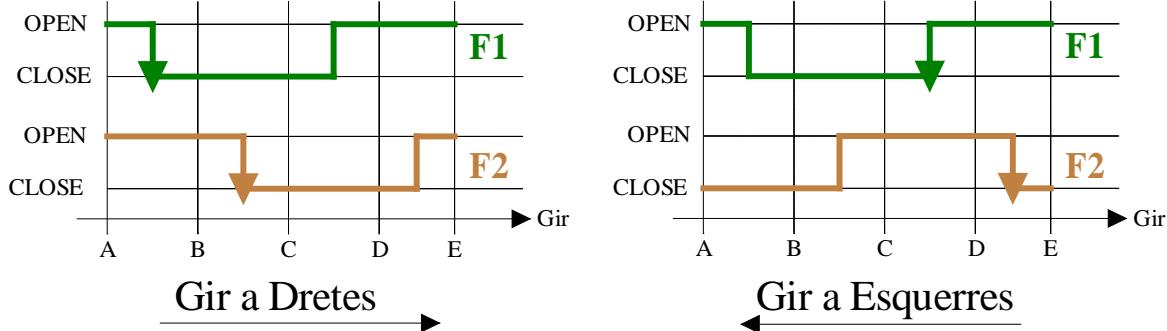


De la figura es dedueix clarament per que es denomina un *encoder* de quadratura. Si considerem com període el gir que cal per repetir un pols, les fases F1 i F2 estan separades  $1/4$  de període. De dos senyals iguals desfasades  $1/4$  de període diem que es troben en quadratura.

Quadratura correspon a un desfasament de  $90^\circ$ , però, en aquest context, es millor no parlar de graus ja que  $90^\circ$  de desfasament es  $1/4$  de mitja volta, que correspon a un gir de  $180^\circ/4 = 45^\circ$  del disc central ja que fa dos polsos per volta.

La figura anterior mostra la seqüència d'estats dels dos interruptors tant si es gira en una direcció com la contraria. Girant a dretes la seqüència és A-B-C-D-E. Girant a esquerres la seqüència és E-D-C-B-A.

Es interessant analitzar la posició dels flancs de baixada (passos de obert a tancat) de les fases F1 i F2 quan es gira en els dos sentits tal i com es mostra a la figura següent:



Si giren a dretes, quan es produeix un flanc de baixada de F1, F2 esta obert i quan es produeix un flanc de baixada de F2, F1 esta tancat. En canvi, si giren a esquerres, quan es produeix un flanc de baixada de F1, F2 esta tancat i quan es produeix un flanc de baixada de F2, F1 esta obert. En definitiva, sabent l'estat del altre canal quan es produeix un flanc d'un canal ens permet saber la direcció del gir.

La manera mes senzilla, i la que farem servir en aquesta pràctica, de fer servir l'*encoder* es comptar polsos només a una de les fases. Si fem servir F1 tindrem la següent taula de comportament:

F1	F2	Comptador
↓	OPEN	+1
↓	CLOSE	-1

Amb això comptarem +1 o descomptarem -1 a cada flanc de F1 dependent de la direcció de gir del *encoder*. Si volguéssim conèixer el gir del *encoder* amb una precisió més gran que l'angle associat a un pols, podríem fer servir tots dos flancs a totes dues fases per obtenir una precisió de posicionament relatiu de 1/4 de pols.

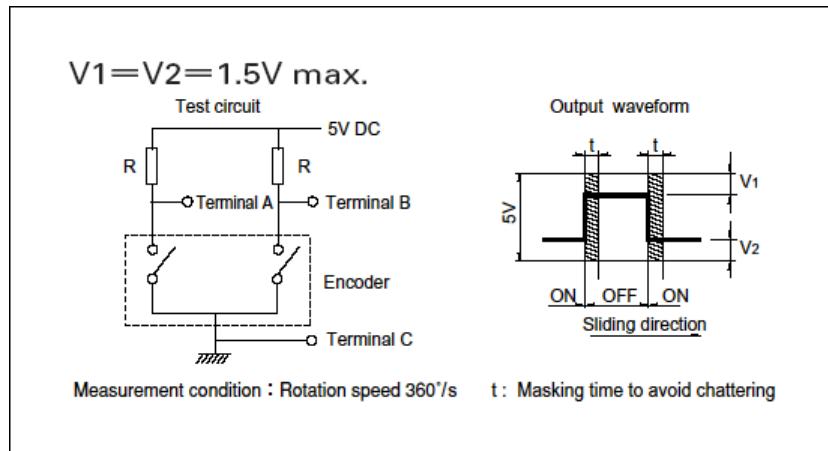
F1	F2	Comptador
↓	OPEN	+1
↑	CLOSE	
CLOSE	↓	
OPEN	↑	
↓	CLOSE	-1
↑	OPEN	
OPEN	↓	
CLOSE	↑	

Com que el nostre *encodernomés* té un punt de detenció per cada pols complet, no te sentit intentar una precisió de 1/4 de període.

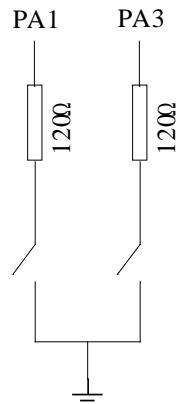
L'encoder de la placa es el model EC12E24204A9 de la companyia ALPS. La seva informació bàsica la podeu trobar en aquesta referència web:

<http://www.alps.com/WebObjects/catalog.woa/E/HTML/Encoder/Incremental/EC12E/EC12E24204A9.shtml>

La dada principal que hem de saber, però, es que té 24 punts de parada i el seu esquema de connexions:



La connexió de l'encoder al microcontrolador es troba descrita al document "[Esquemes.pdf](#)" i al *board file* i es la que es mostra a la figura següent:



Encoder

Com es veu les dues sortides de l'*encoder* estan connectades a les línies GPIO PA1 i PA3 a través de dues resistències de 120Ω. Aquestes resistències son una protecció pel microcontrolador pel cas de que erròniament definíssim PA1 o PA3 como GPIO de sortida. A efectes pràctics es com si fossin curtcircuits.

L'encoder, com es pot veure només força el "0", per tant, ens caldrà fer servir entrades GPIO amb *pull-up* per PA1 i PA3 per disposar d'un "1" quan l'interruptor està obert.

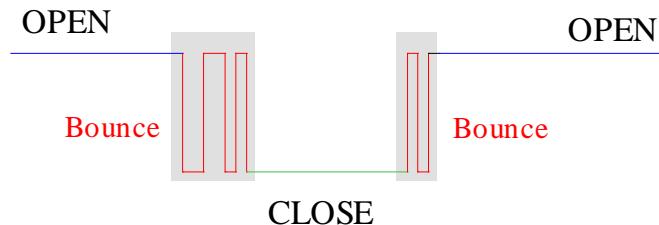
Com que els interruptors estan connectats a massa, el pas d'obert a tancat es un **flanc de baixada**, tal i com es mostra a les figures anteriors. En tot cas, com que això només afecta a si es compta o descompta al girar a dretes o esquerres, no ens afectarà de manera important.

### C3.2 Gestió del'encoder amb interrupcions

La millor manera de gestionar l'*encoder*és fent servir interrupcions. Definirem una variable global **count** de tipus volatile int32\_t que variarà en un valor +1 o -1 a cada flanc de baixada(pas a tancament del interruptor) d'una de les fases.

Una manera de treballar seria programar les interrupcions de PA1 per flanc de baixada i, a dins de la RSI d'aquesta interrupció incrementar o decrementar el comptador dependent del estat de PA3. El problema de fer servir aquesta tècnica senzilla son els **rebots**(*Bounce*) dels interruptors del'encoder.

Quan un interruptor canvia d'estat, pot fer diversos canvis fins que es fixa en el valor final.



En el nostre cas, això vol dir que a cada transició del canal del'encoder associat a PA1, tant si es de pujada com de baixada, es comptarà o descomptarà varies vegades i el sistema no serà fiable.

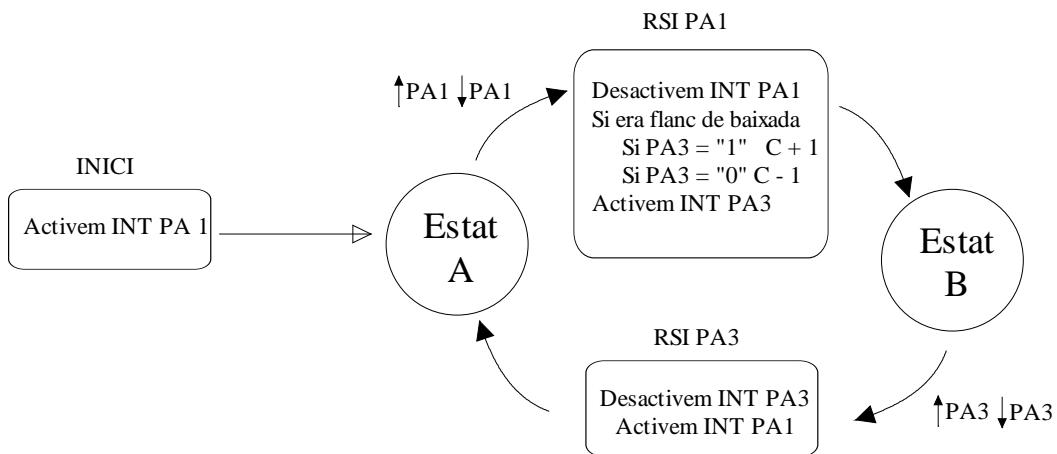
El que hem de fer, per evitar el problema dels rebots, es comptar només el primer flanc de cada transició de pujada o baixada.

Sabem que els senyals de les dues fases F1 (PA1) i F2 (PA3) estan en quadratura, per tant, després d'un flanc d'una de les fases i abans de tenir un altre flanc a la mateixa fase ha de venir un flanc de l'altra fase.

Programarem interrupcions amb el flanc de pujada i baixada dels dos canals PA1 i PA3, però només habilitarem les interrupcions d'un dels dos canals. El sistema podrà estar doncs en dos estats:

- Estat A: Esperant interrupcions a PA1
- Estat B: Esperant interrupcions a PA3

La següent figura mostra el sistema proposat:



En arrancar el sistema, únicament habilitarem les interrupcions de PA1. El programa correrà en l'estat A, esperant una interrupció a PA1. Quan es doni un flanc de baixada o pujada a PA1, s'executarà la seva RSI. Dins d'aquesta:

- Es deshabilitaràn les interrupcions a PA1
- En cas de que el flanc fos de baixada (es a dir PA1 baixa)
  - S'augmentarà el comptador (+1) si PA3 es alta
  - Es disminuirà el comptador (-1) si PA3 es baixa
- S'habilitaràn les interrupcions a PA3

En retornarà al programa principal que ara correrà al estat B, esperant una interrupció a PA3. Quan es doni un flanc de baixada o pujada a PA3, s'executarà la seva RSI. Dins d'aquesta:

- Es deshabilitaràn les interrupcions a PA3
- S'habilitaràn les interrupcions a PA1

Fent servir aquest procediment, només es capturarà el primer flanc de cada transició de cada fase ja que es desactiva la pròpia interrupció i no es torna a activar fins que no s'ha donat un flanc a la fase complementària.

### **C3.3Realització del codi**

Per implementar la gestió del'encoder ens caldran tres elements de codi:

- Inicialització dels GPIO del'encoder i les interrupcions
- Rutines de servei d'interrupció
- Programa principal

Per fer la inicialització del'encoder desenvoluparem una funció **initEncoder**, aquesta inicialització és semblant a la proposada a la pràctica P4 pel cas del polsador d'usuari. Podeu tornar a consultar aquesta pràctica si en teniu dubtes. En tot cas, dins d'aquesta funció, haurem de realitzar les següents accions:

- Configurar les línies GPIO PA1 i PA3 com entrades digitals amb *Pull-Up*. La informació sobre la programació del ports, i en concret, la programació en mode d'entrada amb *pull-ups*'ha explicat a la pràctica P2, en concret al apartat P2.4. De fet, aquest punt es opcional ja que es la configuració per defecte després de cridar *baseInit( )*.
- Programar el *System configuration controller* (SYSCFG), en concret el registre EXTICR1 (SYSCFG->EXTICR[0]), per configurar les entrades EXTI1 i EXTI3 a les línies dels ports PA1 i PA3.  
⚠️ No oblideu de habilitar el rellotge de SYSCFG que es troba a APB2
- Programar l'*External interrupt controller* (EXTI) per tal que les interrupcions a les línies EXTI1 i EXTI 3 es configuri com actives tant per flanc de pujada com per flanc de baixada. Per fer ho:
  - Activarem la generació d'interrupcions als canals 1 i 3 per flanc de pujada amb el registre EXTI->RTSR
  - Activarem la generació d'interrupcions als canals 1 i 3 per flanc de baixada amb el registre EXTI->FTSR
  - Esborrarem el flag d'interrupció pendent tant als canals 1 i 3 amb el registre EXTI->PR
  - Habilitarem el canal 1 amb EXTI->IMR. Recordeu que, quan arrenca el sistema només s'han d'habilitar les interrupcions de PA1.
- Programar el *Nested vectored interrupt controller* (NVIC) per activar les línies de petició d'interrupció dels canals EXTI1 i EXTI3 que tenen noms **EXTI1\_IRQn** i **EXTI3\_IRQn** respectivament.

EP1 Escriviu una versió preliminar de la funció <b>initEncoder</b> .
--

Una vegada definida la funció **initEncoder**, hem de desenvolupar les RSI associades a les línies PA1 i PA3. Aquestes interrupcions estan associades als vectors **EXTI1\_IRQHandler** i **EXTI3\_IRQHandler** respectivament.

Les funcions a realitzar dins de cada RSI estan descrites a la figura del apartat C3.2. El format que han de tenir les funcions RSI està descrit a la pràctica P4, per tant, no es repetirà aquí.

Les funcions a realitzar per l'RSI de PA1 son les següents:

- Inhabilitar les interrupcions a PA1 mitjançant EXTI->IMR
- Esborrar el *flag* d'interrupció PA1 pendent mitjançant EXTI->PR
- En cas de que sigui un flanc de baixada (bit PA1 baix)
  - Incrementar la variable **count** en +1 si PA3 es alt
  - Reduir la variable **count** en -1 si PA3 es baix
- Habilitar les interrupcions a PA3 mitjançant EXTI->IMR
- Esborrar el *flag* d'interrupció PA3 pendent mitjançant EXTI->PR

En el cas de la RSI de PA3 les funcions son similars, però no s'actua sobre el comptador, per tant les accions a realitzar seran:

- Inhabilitar les interrupcions a PA3 mitjançant EXTI->IMR
- Esborrar el *flag* d'interrupció PA3 pendent mitjançant EXTI->PR
- Habilitar les interrupcions a PA1 mitjançant EXTI->IMR
- Esborrar el *flag* d'interrupció PA1 pendent mitjançant EXTI->PR

**EP2** Escriviu una versió preliminar del codi de les dues **RSI**.

Una vegada fet el codi de les dues interrupcions només ens manca fer un programa principal **main()** que inicialitzi l'**LCD** i l'**encoder** i que mostri periòdicament al **LCD** el valor de la variable **count**.

- ⚠ No oblideu que la variable **count** s'ha de declarar de tipus **int32\_t** amb modificador **volatile** dins del mòdul **encoder.c**. Per poder accedir a aquesta variable dins de main.c, s'ha de tornar a declarar dins d'aquest fitxer però, aquesta vegada, amb el modificador **extern** afegit. D'aquesta manera s'indica al **linker** que la definició d'aquesta variable es troba a un altre mòdul.

## C3.4 Prova del codi

### Inici del treball de laboratori

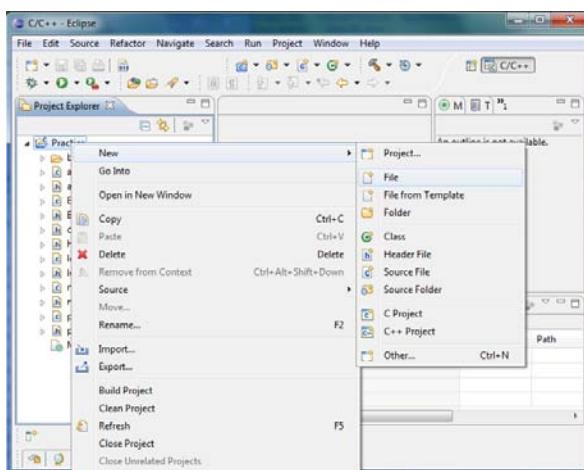
En aquest punt s'inicia el treball de laboratori. Això no vol dir que no hi hagin més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

- ⚡ Abans que res, comproveu el bon funcionament de la placa.

Per desenvolupar el codi de gestió del *encoder*, el podeu incorporar als fitxers que ja teniu, però seria recomanable que generéssiu dos nous fitxers **encoder.c** i **encoder.h**.

Per crear nous fitxers dins del projecte, seleccioneu el projecte Practica i obriu el menú contextual amb el botó dret del ratolí. Seleccioneu **New → File**



Genereu d'aquesta manera els dos fitxers **encoder.c** i **encoder.h**. Per incorporar dins del procés de compilació el fitxer **encoder.c**, haureu de modificar el fitxer **makefile** com ja hem fet altres vegades a les pràctiques anteriors.

- 💻 Introduïu el codi de la funció **initEncoder**

Introduïu el codi de les dues **RSI** dels canals **EXTI1** i **EXTI3**

Feu un programa que mostri per pantalla de manera periòdica el valor de la variable **count**.

Comproveu que el valor mostrat canvia correctament quan girem el control del *encoder*.

- 💡 Podeu modificar les *RSI* per que s'actualitzi el valor de la variable **count** amb els quatre flancs de les dues sortides del *encoder* tal i com es descriu a C3.1. A cada moviment a la següent posició de parada del control del *encoder* el valor del comptador s'haurà d'incrementar o reduir en 2 unitats.

# Pràctiques amb RTOS

---

Aquestes pràctiques són obligatòries pels estudiants de **SEBM**.

Els estudiants de **SBM**, si hi tenen interès, les poden fer com a pràctiques opcionals després que ho aprovi el seu professor de pràctiques.

A les pràctiques anteriors hem desenvolupat algunes aplicacions sobre el microcontrolador STM32F407 treballant de manera directa sobre aquest. En aquest bloc de pràctiques treballarem amb el sistema operatiu ChibiOS/RT.

# R1 - Introducció a ChibiOS/RT

En aquesta primera pràctica es vol introduir el sistema operatiu ChibiOS/RT que és un més dins la categoria de sistemes operatius en temps real. Veurem com podem executar processos concurrents. També veurem la capa HAL (Hardware Abstraction Layer) que ens permet accedir als perifèrics del microcontrolador d'una manera més senzilla.

## Objectius de la pràctica:

- Aprendre a crear varis processos simultanis

## Estudis Previs:

- Primera versió del programa demanat a R1.3

## Resultats:

- Programa demanat a R1.3 que fa servir 4 processos independents

## R1.1 ChibiOS/RT

No és fàcil definir el que és ChibiOS/RT sense abans definir el que es un RTOS. I no es fàcil definir un RTOS sense definir abans el que es un Sistema Operatiu.

## Sistemes Operatius

Per fer treballar un programa dins d'un microprocessador (o microcontrolador) no cal tenir un sistema operatiu. Capdels programes que hem desenvolupat a la primera part de les pràctiques no fan servir sistema operatiu. En rigor, hem compilat incloent ChibiOS/RT però, de fet, no l'hem fet servir llevat de les rutines d'inicialització i les funcions **DELAY\_US** i **SLEEP\_MS** i, aquestes funcions, no necessiten pas un sistema operatiu per treballar.

Quan una aplicació és poc complexa i els perifèrics que fa servir són senzills, podem escriure tot el codi que cal per l'aplicació. Quan les aplicacions es fan més complexes acostumen a passar dues cosses:

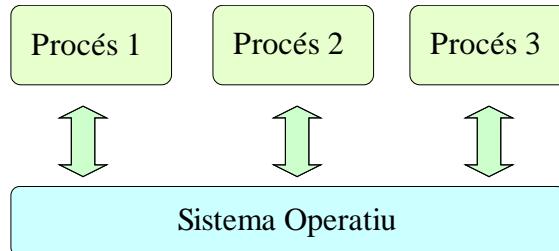
- Cal concorrència
- Cal accedir a perifèrics més complexos

Concorrència vol dir que cal fer varies cosses a la vegada. Fent servir interrupcions podem fer una mica de concorrència, però tenir feines concurrents independents demana, al final, disposar de processos o fils (*Processes / Threads*). Un procés es l'abstracció de la feina més o menys seqüencial que fa un processador. Dit de manera senzilla, fer córrer un programa. Tenir dos processos a l'hora vol dir que hi ha dos programes corrent al mateix temps. En sistemes com el nostre amb una única

CPU això es fa alternant el temps de CPU entre els dos processos. El punt important és que el codi de cada procés s'escriu com si fos l'únic.

Tota això és còmode des del punt de vista de la programació per què podem, per exemple, tenir un programa per escriure l'LCD, un altre per llegir el teclat i un altre per comunicar-se per un port RS-232 i fer-los córrer tots a la vegada.

El problema és que algú s'ha d'encarregar de distribuir el temps de CPU entre els processos. Aquest algú que es troba per sobre dels processos i reparteix el temps de CPU és el sistema operatiu. El sistema operatiu s'encarrega també de gestionar la comunicació entre processos.



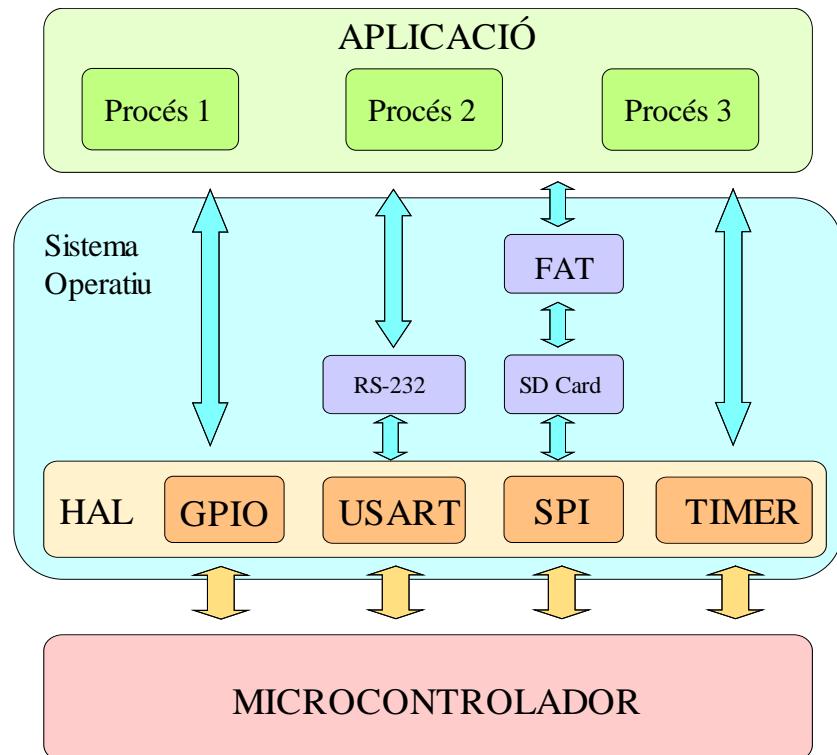
A la figura es mostra un exemple de 3 processos corrent al mateix temps. Els tres processos treballen independentment però tots tres es comuniquen amb el sistema operatiu. Si un procés vol comunicar-se amb un altre, ho ha de fer amb eines del sistema operatiu.

- ⓘ En rigor, no és el mateix parlar de procés (*Process*) que de fil (*Thread*). Normalment els **processos** es consideren bastant independents entre sí, com és el cas de les diferents aplicacions que es poden executar en un PC. Els **fils**, en canvi, són els diferents elements de codi concurrent més relacionats que s'executen dins d'un únic procés. En sistemes basats en microcontrolador que executen una única aplicació la diferència entre procés i fil no es tan clara i per tant, de moment, no farem diferència entre els dos conceptes.

Veurem ara el problema dels perifèrics. Quan un perifèric és senzill podem accedir a ell directament fent servir els seus registres associats. Aquest es el cas dels ports **GPIO**.

Quan el perifèric és més complex, com és el cas del port **SPI** que permet llegir l'acceleròmetre, cal fer funcions per separar l'accés al perifèric de l'ús que es fa d'aquest accés. Aquestes funcions les podem agrupar en mòduls de codi, com és el cas de **accel.c/accel.h** o, a un nivell més alt, podem desenvolupar **llibreries** de codi. Per sistemes complexos, l'accés al hardware pot ser una part important de tot el codi.

Per simplificar la comunicació amb el hardware, un sistema operatiu pot incloure un **HAL** (*Hardware Abstraction Layer*). Aquest element es un conjunt de funcions que fa de pont entre la nostra aplicació i el hardware. Això permet accedir als perifèrics sense saber la implementació concreta que fa el microcontrolador. El sistema operatiu pot tenir altres funcions que accedeixen al hardware mitjançant el propi HAL. A mode de exemple, l'**HAL** pot contenir funcions d'accés a un bus SPI. Les targes SD poden ser connectades a un bus SPI, per tant, el sistema operatiu pot tenir funcions per accedir a una tarja SD. A sobre d'aquestes funcions podem afegir funcions per diferents tipus de sistemes d'arxius com ara FAT.



A la figura es mostra un exemple de sistema que té una aplicació amb 3 processos. L'HAL, que es part del Sistema Operatiu, fa la comunicació de baix nivell amb el hardware, vindria a ser l'equivalent de la BIOS a un PC. A sobre del'HAL, el sistema operatiu pot incloure codi que s'estengui per sobre de la funcionalitat del'HAL. Tal és el cas de afegir el protocol RS-232 sobre l'HAL de la USART o afegir el sistema d'arxius i la gestió d'una tarja SD sobre l'HAL del bus SPI.

Si la nostra aplicació no es comunica directament amb el hardware sinó només amb l'HAL o qualsevol cosa que pengi d'ell, serà portable a altres processadors sempre i quan la definició de com fer servir l'HAL no canviï.

L'accés al hardware quan la nostre aplicació té diferents processos dona lloc a un nou problema. Si dos processos volen accedir al mateix element de hardware, només ho pot fer un (accés exclusiu), i l'altre s'ha d'esperar a que quedi lliure. El sistema operatiu s'ha d'encarregar de l'accés ordenat al hardware. De fet, no només al hardware, qualsevol element que pugui ser accedit per dos processos, sigui hardware, una posició de memòria, el que sigui, s'ha de gestionar ordenadament.

A part de tot l'anterior, el sistema operatiu pot incloure encara molt codi com, per exemple:

- Gestió de l'arrencada de la CPU
- Gestió de la memòria RAM
- Gestió de les interrupcions

En definitiva, i com a resum, un sistema operatiu és un conjunt de codi que s'encarrega de gestionar els recursos de hardware i proporciona serveis comuns per les aplicacions. En el cas de sistemes operatius multiprocés, s'encarrega de repartir el temps entre els diferents processos i proveir de les eines necessàries per la comunicació dels processos entre sí i amb qualsevol element comú.

De sistemes operatius n'hi ha una gran varietat. No tots els sistemes operatius disposen d'**HAL**, ni tots els sistemes operatius permeten variar processos, per tant, és difícil entrar en detall sense parlar d'algún tipus particular de sistema operatiu.

## Sistemes Operatius en Temps Real (RTOS)

Un Sistema Operatiu en Temps Real (RTOS) és un sistema operatiu dissenyat per treballar en un sistema en **temps real**. Un sistema en temps real és un sistema que ha de ser capaç de respondre als successos complint requisits de temps estrictes.

Un exemple de sistema en **temps real** seria un sistema antiincendis que hagi de garantir que entre el moment que la temperatura d'un sensor pugi de 80°C i s'encengui una alarma no han de passar més de 2 segons. Passi el que passi el sistema ha de garantir que això serà així. Els sistemes de control de motors o de gestió de successos a un automòbil com l'ABS, Airbag, etc.. també acostumen a ser sistemes en temps real.

No cal sempre un sistema operatiu per fer un sistema en temps real, però si un sistema en temps real té un sistema operatiu, aquest ha de ser un RTOS. Els RTOS donen garanties estrictes del temps que es triga en saltar d'un procés a un altre (temps de canvi de context), respondre una interrupció, etc...

Els sistemes operatius que es fan servir als PCs, com és el cas de Windows i Linux no poden garantir un temps de resposta i, per tant, no són RTOS. Tampoc és que importi massa per aplicacions d'ofimàtica.

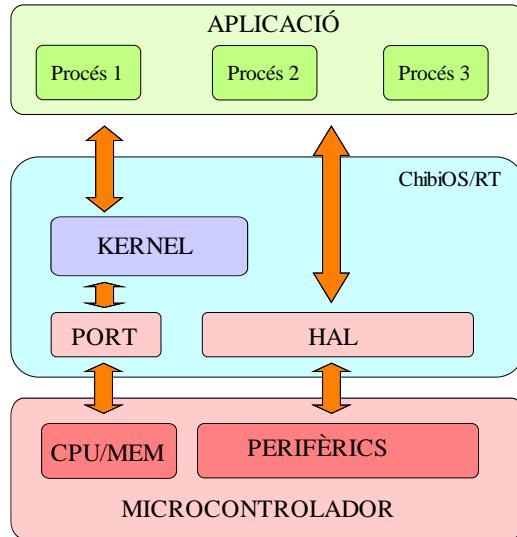
És important destacar que fer servir un RTOS no fa que el nostre sistema sigui a **temps real**. La resta del codi del sistema s'ha de dissenyar també de manera acurada per complir els requisits de temps demanats.

## ChibiOS/RT

[ChibiOS/RT](#) és un RTOS de codi obert dissenyat per treballar amb aplicacions de temps real que requereixen codi compacte i execució eficient amb breu temps de canvi de context. ChibiOS/RT es pot executar en diferents [plataformes](#) incloent les basades en ARM Cortex M4 com és el cas del microcontrolador STM32F407 que fem servir.

ChibiOS/RT té una estructura basada en un Nucli (*Kernel*) envoltat per una capa d'abstracció de hardware (HAL). Tant el nucli com l'HAL són modulars, és a dir, podem compilar només les parts del sistema operatiu que fem servir de manera que el codi ocupa el mínim d'espai.

La següent figura mostra l'estructura bàsica d'un sistema basat en ChibiOS/RT



Els microcontroladors, com ja sabem, són dispositius que inclouen CPU, memòria i perifèrics. La part de ChibiOS/RT que es comunica amb la CPU i la memòria es diu PORT. La part que es comunica amb els perifèrics es l'HAL. El nucli (Kernel) es troba per sobre del PORT i és, per tant, independent de la CPU que fem servir. Aquesta estructura modular fa relativament senzill portar el sistema operatiu a un altre microcontrolador ja que no cal reescriure el nucli.

Com ja hem comprovat a les pràctiques anteriors, no cal fer servir l'HAL per accedir al perifèrics del microcontrolador. Per tant, podem, si volem, compilar només el PORT de la nostra CPU i els elements del nucli que farem servir.

A continuació es descriuen els elements més importants:

**PORT:** Accés de baix nivell a la CPU i la memòria

- Suport pel Nucli
- Accés al'NVIC (Gestor d'interrupcions)
- Inicialització del processador

**KERNEL:** Part principal del sistema operatiu

- Serveis bàsics de gestió de sistema i processos
- Sincronització
- Gestió de memòria
- Gestió de fitxers

**HAL:** Accés al hardware

- Configuració general
- Un *driver* per cada perifèric

A part d'això ChibiOS/RT també conté funcions per inicialitzar els perifèrics bàsics de plaques concretes com la placa **STM32F4 Discovery** que fem servir. És per això que no cal programar els GPIO dels leds ni del pulsador d'usuari de la placa.

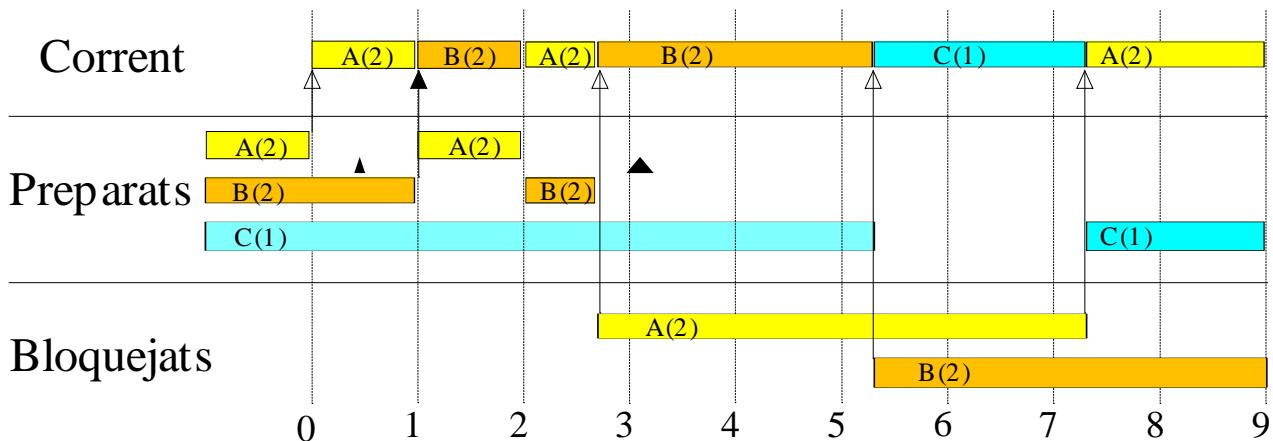
## **R1.2Processos**

Dins de ChibiOS/RT podem tenir tants processos com hi càpiguen amb els recursos disponibles del processador. Cada procés té una prioritat assignada.

Els processos poden estar corrent (*Running*), preparats per córrer (*Ready*) o bloquejats (*Blocked*). Per què un procés estigui preparat per córrer no ha de estar bloquejat per cap mecanisme de sincronització. De tots els processos del sistema només un pot córrer en un moment donat ja que només hi ha una CPU.

El responsable de repartir el temps de CPU entre processos és el denominat *Scheduler*. Aquest element del sistema operatiu assigna el temps de CPU al procés que tingui més prioritat. Si hi ha més d'un procés amb la mateixa prioritat i tota la resta està per sota, aquests processos es repartiran el temps disponible.

Es molt important notar que la jerarquia de prioritats es compleix estrictament. Si sempre hi ha un procés amb més prioritat no bloquejat, els que tinguin menys prioritat **no correran mai**.



A mode de exemple, a la figura es mostra un sistema que arrenca a temps 0 amb tres processos. A i B tenen prioritat 2 i C té prioritat 1. El sistema té un tic de rellotge que, per defecte, es dona cada mil·lisegon i que es fixa amb la variable **CH\_FREQUENCY** que es troba dins de **chconf.h**.

Quan arrenca el sistema, dels tres processos preparats, n'hi ha dos, A i B que tenen prioritat per sobre del'altre, C. Aquests dos es reparteixen el temps alternativament amb un tic de rellotge per cadascú en el cas del nostre exemple. Entre el tic 2 i el 3 el procés A es bloqueja, per exemple, per que està esperant un senyal extern. Ara només el procés B té prioritat màxima i omple tot el temps de CPU. Poc després del tic 5, el procés B es bloqueja també. Ara pot córrer el procés C però només fins el moment en que es desbloqueja el procés A després del tic 7 donat que, quan ho fa, és el de més prioritat.

❶ En realitat, quan dos processos tenen la mateixa prioritat i no hi ha processos amb menys prioritat preparats per córrer, es reparteixen el temps però no necessàriament amb un tic de rellotge per cadascun. El nombre de tics disponible per cada procés abans de que li toqui córrer al' altre es diu *time quantum* i s'especifica amb la variable **CH\_TIME\_QUANTUM** que es troba dins del fitxer **chconf.h** i que per defecte es de 20 tics, és a dir, 20ms amb el tic per defecte d'1ms.

Si un procés està corrent i no vol fer servir tot el seu *time quantum* disponible, pot renunciar al temps que li queda cridant la funció **chThdYield**.

Si fixem la variable **CH\_TIME\_QUANTUM** a zero, el sistema es torna cooperatiu per processos amb la mateixa prioritat. Es a dir, no es fa servir el *time quantum* i l'ús de CPU només pot passar d'un procés a un altre amb la mateixa prioritat si es fa servir la funció **chThdYield**.

Tot això no afecta a processos amb més prioritat que sempre interrompran altres processos amb menys prioritat en el moment just en que estiguin preparats per córrer.

## Gestió de la memòria de pila

A cada procés, en el moment de crear-lo, se li assigna una certa quantitat de memòria. Aquesta memòria es fa servir per tres funcions:

- Guardar informació del propi procés que fa servir el sistema operatiu
- Guardar informació de context
- Pila del procés

La informació del procés permet identificar el nom, la prioritat i altres informacions al sistema operatiu. La informació de context permet emmagatzemar l'estat que tenia el procés just quan el *Scheduler* el treu del estat de córrer (*Running*). D'aquesta manera el procés pot continuar sense problemes quan el *Scheduler* li torna a donar temps de CPU.

Finalment tenim la pila. En un sistema amb CPU la pila és el lloc on s'emmagatzemen els paràmetres de crida a funcions, la direcció i els valors de retorn de funcions i les variables dinàmiques (totes les que es defineixen dins de les funcions sense el modificador *static*).

Com que cada procés té una pila independent i no ens interessa assignar molta memòria a cada procés per no quedar-nos sense, s'ha de vigilar molt amb l'ús que es fa de les variables dinàmiques i de les crides anellades a funcions (Una funció que crida a un altre i aquesta a un altre i així moltes vegades). Una manera d'alleugerir aquest problema és declarar les variables com estàtiques, però és important recordar que això fa, generalment, que les funcions no siguin reentrants.

### **R1.3 Exemple de codi**

#### **Inici del treball de laboratori**

En aquest punt s'inicia el treball de laboratori. Això no vol dir que no hi hagi més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

- ⚡ Abans que res, comproveu el bon funcionament de la placa.
- 💻 Obriu el fitxer "**Practica-R1.zip**", agafeu els fitxers **process.c** i **process.h** i fiqueu-los dins del vostre directori de treball "**Practica**". Incloeu el fitxer **process.c** dins de la llista de fitxers que han de compilarse mitjançant la modificació del fitxer **makefile** tal i com ho hem fet ja altres vegades.

Veurem ara un exemple senzill amb dos processos implementat amb la funció **test2threads(void)**. Un primer procés serà el programa principal, dins del qual, generarem un procés fill que, a partir de la seva creació, treballarà de manera autònoma. Després de crear el procés fill, el programa principal farà pampallugues al led blau per indicar que està corrent.

El procés fill, un cop engegui, també farà pampallugues, però amb el led taronja. Per tant, mirant quin led fa pampallugues podem deduir quin procés (programa principal o procés fill) està en marxa.

Estudiarem, ara, el contingut del fitxer **process.c**

Quan arrenca el sistema operatiu, després de cridar **baseInit( )**, només hi ha un procés en actiu que té una prioritat **NORMALPRIORITY**. Qualsevol procés addicional serà fill d'aquest procés inicial. El valor numèric de la prioritat **NORMALPRIORITY** no és important ja que fixarem la prioritat de tots els processos fills de manera **relativa** a la del programa principal. La mínima prioritat que podem fer servir a un procés es **LOWPRIORITY** i la màxima **HIGHPRIORITY**. **NORMALPRIORITY** es troba a un punt entremig entre aquestes prioritats.

El següent que fa **test2threads** després d'arrencar el sistema és crear un procés/fil fill:

```
// Child thread creation  
chThdCreateStatic(waChild, sizeof(waChild), NORMALPRIORITY, thChild, NULL);
```

Tota la informació sobre ChibiOS/RT la podeu trobar a:

<http://chibios.sourceforge.net/html/index.html>

En concret, la funció per crear processos **chThdCreateStatic** es troba a:

Modules → Kernel → Base Kernel Services → Threads

```
Thread * chThdCreateStatic ( void *      wsp,
                           size_t       size,
                           tprio_t     prio,
                           tfunc_t     pf,
                           void *      arg
                         )
```

Creates a new thread into a static memory area.

Note:

A thread can terminate by calling chThdExit() or by simply returning from its main function.

Parameters:

[out]	wsp	pointer to a working area dedicated to the thread stack
[in]	size	size of the working area
[in]	prio	the priority level for the new thread
[in]	pf	the thread function
[in]	arg	an argument passed to the thread function. It can be NULL.

Returns:

The pointer to the Thread structure allocated for the thread into the working space area.

Function Class:

Normal API, this function can be invoked by regular system threads but not from within a lock zone.

Definition at line 184 of file chthreads.c.

A ChibiOS/RT el nom de la funció està relacionat amb el bloc de funcions a que pertany. Totes les funcions del nucli (*Kernel*) comencen per **ch**. Les funcions sobre fils/processos continuen amb **Thd**. La funció **chThdCreateStatic**, per tant, s'identifica com una funció del nucli associada als processos. Aquesta funció, en particular, permet crear un nou procés fent servir memòria estàtica, es a dir, assignada en temps de compilació.

A continuació descriurem els paràmetres de la crida a la funció:

**wsp** Punter al'àrea de treball del procés, de tipus punter a **void**

Aquest paràmetre apunta a la memòria del'àrea de treball del nou procés.

Per crear una àrea de treball per un procés, la manera més senzilla és fer servir la macro **WORKING\_AREA**. Aquesta macro reserva una àrea de treball que inclou la quantitat que es demani de bytes per la pila.

Dins del fitxer **processos.c** trobem aquestes dues línies:

```
// Working area for the child thread
static WORKING_AREA(waChild,128);
```

Això vol dir que tenim un àrea de treball pel procés fill **waFill** (procés fill *WorkingArea*) amb 128 bytes de mida de la pila.

**size** Mida del àrea de treball. Si la nostra àrea de treball és **area\_de\_treball**, la seva mida es pot trobar sempre amb:

```
sizeof(area_de_treball)
```

Aquesta mida serà sempre més gran que l'espai associat a la pila, ja que inclou altres estructures a part d'aquesta.

**prio** Prioritat del nou procés/fil a crear.

La millor manera de treballar és fer servir prioritats relatives a la del programa principal que és **NORMALPRIORITY**. Així, farem servir **NORMALPRIORITY+1**, per exemple, si volem una prioritat un nivell per sobre de la del programa principal o **NORMALPRIORITY-1**, si volem una prioritat un nivell per sota de la del programa principal.

① Es molt important tenir en compte que un procés pot esdevenir actiu tan bon punt el creem. Això vol dir que si creem un procés amb més prioritat que el procés que l'ha creat, just en el moment de crear-lo, el procés creador passarà a ser inactiu degut a que el nou procés es més prioritari.

**pf** La funció del nou procés

Aquesta funció serà on arrenqui a treballar el nou procés, i ha de tenir una definició:

```
static msg_t nom_de_funcio(void *arg)
```

Es a dir, una funció que pren com argument un punter a **void** i que pot retornar un valor de tipus **msg\_t**. Un procés acaba completament i desapareix del'arbre de processos si la seva funció retorna. El valor de retorn pot indicar a altres processos el motiu pel qual ha acabat.

Es important ser conscient que una funció no identifica un procés. El que l'identifica es la seva àrea de treball. Podem tenir dos processos independents que son arrencats amb la mateixa funció. En aquests casos el paràmetre d'entrada **arg** es fa servir per particularitzar l'arrencada dels dos processos.

**arg** Paràmetre a enviar a la funció d'arrencada del procés (amb tipus punter a **void**)

Aquest paràmetre serà enviat a la funció que arrenca el procés. D'aquesta manera es possible arrencar dos processos diferents amb una mateixa funció.

Al nostre programa la crida a la creació del procés és:

```
chThdCreateStatic(waChild, sizeof(waChild), NORMALPRIORITY, thChild, NULL);
```

Per tant, es crea un procés amb àrea de treball **waFill** que té mida **sizeof(waFill)**, amb prioritat igual al programa principal, associat a una funció **filFill** a la que no es passa cap argument (**NULL**).

① La documentació original d'ajuda sobre la creació de fils a ChibiOS/RT es troba aquí:

<http://chibios.org/dokuwiki/doku.php?id=chibios:howtos:createthread>

Després de la crida a la funció **chThdCreateStatic**, la funció **test2threads**, té el codi:

```
while (TRUE)
{
    // Turn on blue LED using BSRR
    (LEDS_PORT->BSRR.H.set)=BLUE_LED_BIT;

    // Pause
    busyWait(3);

    // Turn off blue LED using BSRR
    (LEDS_PORT->BSRR.H.clear)=BLUE_LED_BIT;

    // Pausa
    busyWait(3);
}
```

És a dir, s'entra en un bucle infinit en el que es fan pampallugues al led blau.

La funció **thChild** que arrenca el nou fil creat té una estructura similar al bucle infinit de funció **test2threads**. L'únic que fa és fer pampallugues al led taronja i amb una freqüència lleugerament diferent. Tot i que té un codi de retorn, aquesta funció no retorna mai.

Sobre el programa s'han de fer dues consideracions addicionals:

- Els leds blau i taronja es troben al mateix port de GPIO i, per tant, son un recurs compartit per dos processos. En el nostre cas això no pot donar problemes per que les instruccions que fem servir per accedir al port són atòmiques (no divisibles).
  - Tots dos processos criden a la mateixa funció **busyWait**. Per que això no doni problemes aquesta funció ha de ser reentrant. És a dir, ha de poder ser cridada mentre encara està en marxa. Això obliga, entre altres cosses, a no fer servir assignacions estàtiques de memòria dins de la funció. Tota la memòria de la funció es trobarà a la pila del procés que la crida i, per tant, podrem tenir una crida a la funció per cada procés.  
*Podríem tenir més d'una crida per procés si la funció fos recursiva però no és el cas.*
- Definiu una funció **main( )** a **main.c** que només cridi a **test2threads**. Compileu i proveu el codi dins la placa de pràctiques. Els dos leds blau i taronja haurien de fer pampallugues indicant que tots dos processos, el programa principal i el procés fill, estan en marxa.

Degut a que tant el programa principal com el procés fill tenen la mateixa prioritat **NORMALPRIORITY**, tots dos processos es reparteixen el temps de CPU. Evidentment, cada procés rep la CPU un mil·lisegon de temps abans de canviar a l'altre procés.

- Proveu d'assignar al procés fill una prioritat **NORMALPRIORITY+1** en lloc de **NORMALPRIORITY**. Descriu el resultat i expliqueu el motiu pel que es dona aquest resultat.
- Proveu d'assignar al procés fill una prioritat **NORMALPRIORITY-1** en lloc de **NORMALPRIORITY**. Descriu el resultat i expliqueu el motiu pel que es dona aquest resultat.

En un sistema multiprocés és sempre una mala idea el fer estats d'espera fent treballar la CPU. Si un procés fa aquest tipus d'espera, els processos amb menys prioritat no podran fer servir l'espera del procés més prioritari per poder córrer. Es per això que sempre que haguem de fer esperes de més d'un mil·lisegon, convé fer servir la macro **SLEEP\_MS** per fer-les. Aquesta macro el que fa és cridar la funció **chThdSleep** que treu el procés del control de la CPU durant un nombre determinat de tics del rellotge, és a dir, el posa a dormir. D'aquesta manera els altres processos poden córrer mentre un procés més prioritari està dormint.

- Substituïu les crides a **busyWait(3)** per **SLEEP\_MS(300)** i **busyWait(5)** per **SLEEP\_MS(500)**. Torneu a assignar al procés fill una prioritat **NORMALPRIORITY+1** en lloc de **NORMALPRIORITY**. Descriu el resultat i expliqueu el motiu pel que es dona un resultat diferent del cas en que fèiem servir la funció **busyWait**.

En aquest punt hauria de ser clar que en un sistema multiprocés, cada procés, tingui la prioritat que tingui, ha de fer servir la CPU durant tant poc temps com pugui per deixar el màxim de temps de CPU als altres processos. Un procés que no pot córrer en un moment donat, encara que no hi hagi processos més prioritaris, direm que està **bloquejat**. En el nostre exemple, un procés que crida a **SLEEP\_MS** es bloqueja a si mateix durant el temps indicat en la crida.

Aquesta no és l'única manera en que es pot bloquejar un procés. Un procés es pot bloquejar també, per exemple:

- Esperant a que acabi de fer una feina un altre procés
- Esperant una interrupció
- Esperant que un element d'ús comú quedí lliure

Tots aquests mecanismes entren dins de la categoria de sincronització de processos i els veurem en altres pràctiques.

Per acabar aquesta primera pràctica sobre ChibiOS/RT demanarem ampliar en nostre programa:

- Modifiqueu el contingut de **process.c** (i **process.h** si cal) per tenir 4 processos en marxa al mateix temps. Els dos primers processos són el programa principal que controla el led **blau** i el procés que ja hem fet servir que controla el led **taronja**. Els dos nous processos han de controlar els altres dos leds **verd** i **vermell** amb períodes diferents dels altres, com per exemple

400ms i 700ms. Tots els processos s'han de generar dins de la mateixa funció principal que cridarem des de **main()**.

**EP1** Feu una primera versió de com podem fer el programa que es demana al punt anterior.

💡 Per crear els tres processos des de la funció principal, la manera més senzilla és fer crides a tres funcions diferents i que cadascuna controli un led. Donat que els tres processos fan una feina semblant, podem definir una única funció i crear cada procés amb la mateixa funció d'arrencada. L'assignació de feines diferents a cada procés es faria canviant l'argument amb que es crida a la funció (últim paràmetre de **chThdCreateStatic**) que, obviament, ja no seria NULL.

Si voleu acabar de compactar el codi, fins i tot el programa principal pot fer servir la funció d'arrencada dels altres fils per fer les seves pròpies pampallugues al led blau.

## R2 - Compartició d'elements comuns

En aquesta segona pràctica sobre sistemes operatius es vol introduir el concepte d'elements comuns d'ús exclusiu que només un procés pot fer servir a l'hora. Per poder arbitrar els accessos als elements comuns definirem i farem servir un elements de sincronització de processos denominats **Mutex**.

### Objectius de la pràctica:

- Entendre el problema associat a l'accés a elements comuns en un entorn multiprocés
- Resoldre els problemes de sincronització amb semàfors i mutexes

### Estudis Previs:

- EP1 : Feu una primera versió del codi demanat a R2.2
- EP2 : Feu una primera versió del codi demanat a R2.4

### Resultats:

- Programes demanats als diferents apartats

### R2.1 Introducció als Semàfors

En un sistema amb múltiples processos acostuma a passar que hi ha elements comuns que demanen accés exclusiu, és a dir que, en un moment determinat, només hi pot accedir un procés.

A mode d'exemple tenim l'LCD de la nostra placa. Per escriure al'LCD s'ha de generar una seqüència d'ordres. Si dos processos intenten accedir a l'hora al'LCD, les seqüències es barrejaran i, tard o d'hora, es produirà un conflicte i la informació al LCD esdevindrà incorrecta.

Un altre exemple seria el cas d'una variable o matriu de dades compartida. Si dos processos intenten accedir, en mode escriptura, a les mateixes dades al mateix temps, pot donar-se un conflicte que corrompi les dades.

### Semàfors binaris

ChibiOS/RT, com qualsevol sistema operatiu multiprocés, proporciona eines de sincronització bàsica en forma de semàfors binaris.

Un semàfor binari es un element de sincronització que només pot tenir dos estats: **Illiure** (*Not taken*) o **ocupat** (*Taken*). Qualsevol procés pot intentar agafar un semàfor. Si està **Illiure**, el semàfor passarà al estat d'ocupat i el procés continuarà treballant. S'entén que aquest procés ha agafat aquest semàfor. Si quan un procés intenta agafar un semàfor aquest es troba **ocupat**, el procés quedarà bloquejat a l'espera que el semàfor quedí **Illiure**.

Els semàfors binaris es defineixen mitjançant una variable de tipus **BinarySemaphore**.

Un exemple de definició d'un semàfor seria:

```
BinarySemaphore    semafor_1;
```

On es definiria un semàfor binari de nom **semafor\_1**

Les funcions principals per treballar amb semàfors son:

```
void chBSemInit ( BinarySemaphore * pbsm , taken )
```

Inicialitza la variable associada a un semàfor indicant el seu estat inicial que pot ser lliure taken=**FALSE** o ocupat taken=**TRUE**. Aquesta funció només es crida al principi del programa per iniciar el semàfor.

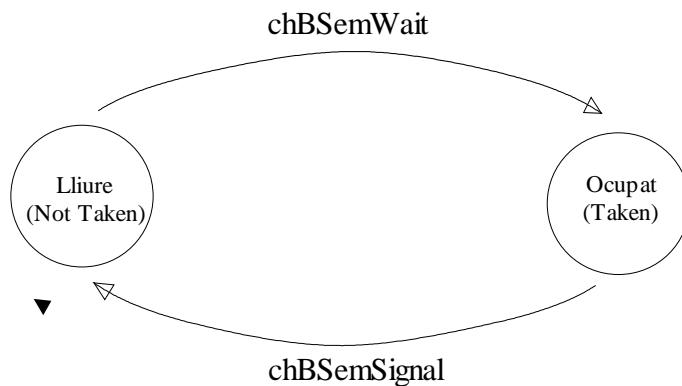
```
msg_t chBSemWait ( BinarySemaphore * pbsm )
```

Intenta agafar un semàfor. Si està lliure, el procés continuarà. Si no, el procés quedarà bloquejat fins que el semàfor quedi lliure, llavors l'agafarà i continuarà.

Normalment aquesta funció retorna **RDY\_OK**, però si es fa un reset del semàfor retornarà **RDY\_RESET**.

```
void chBSemSignal ( BinarySemaphore * pbsm )
```

Deixa lliure un semàfor. Si està lliure, quedarà igual. Si no, passarà d'estar ocupat a estar lliure i un altre procés el podrà agafar. Aquesta funció mai bloqueja al procés.



Hi ha més funcions que permeten, entre altres funcionalitats, esperar un semàfor durant un temps o fer *reset* dels semàfors. Tota la informació sobre semàfors binaris a ChibiOS/RT es troba a:

[http://chibios.sourceforge.net/html/group\\_\\_binary\\_\\_semaphores.html](http://chibios.sourceforge.net/html/group__binary__semaphores.html)

Els semàfors binaris son una potent eina de sincronització entre processos com veurem a continuació.

## **R2.2Exemple d'ús de Semáfors**

### **Inici del treball de laboratori**

En aquest punt s'inicia el treball de laboratori. Això no vol dir que no hi hagin més estudis previs, només vol dir que abans no es demana cap interacció amb l'ordinador ni la placa.

Abans d'anar a la sessió de laboratori heu de llegir la resta del manual de la pràctica i fer els estudis previs que es demanen.

- ⚡ Abans que res, comproveu el bon funcionament de la placa.
- 💻 Obriu el fitxer "**Practica-R2.zip**", agafeu els fitxers **mutexes.c** i **mutexes.h** i fiqueu-los dins del vostre directori de treball "**Practica**". Incloeu el fitxer **mutexes.c** dins de la llista de fitxers que han de compilar-se mitjançant la modificació del fitxer **makefile**.

Fixeu-vos en la primera part del fitxer **mutexes.c** on es troba la funció **semaphoreExample**

```
// Binary semaphore
BinarySemaphore    semaf;

// Working area for the semaphore child thread
static WORKING_AREA(waSem,128);

// Child thread function prototype
static msg_t thSem(void *arg);

// Process synchronization example that uses semaphores
void semaphoreExample(void)
{
    // Global initialization
    baseInit();

    // Initializes the semaphore as not taken
    chBSemInit(&semaf, FALSE);

    // Creates a child thread
    chThdCreateStatic(waSem, sizeof(waSem), NORMALPRIO+1, thSem, NULL);

    while(1)
    {
        SLEEP_MS(300);           // Wait 300ms
        chBSemSignal(&semaf);   // Send signal to child
    }
}
```

Veiem que hem definit un semàfor binari de nom **semaf** i un àrea de treball **waSem** per un subprocés.

La funció **semaphoreExample** comença amb la inicialització del sistema i segueix amb la inicialització del semàfor en estat de lliure. A continuació creem un fil fill amb funció d'arrencada **thSem** i prioritat un punt per sobre del programa principal. Finalment passem a un bucle infinit en el que cada 300ms deixem lliure el semàfor amb la funció **chBSemSignal**.

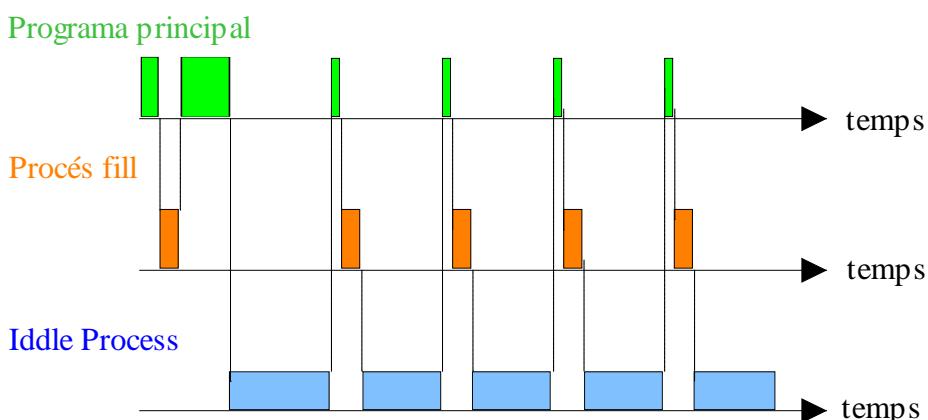
El procés fill definit per la funció d'arrencada **thSem** té el següent codi:

```
// Child thread that changes the state of the orange LED
// at each semaphore synchronization
static msg_t thSem(void *arg)
{
    while(1)
    {
        chBSemWait(&semaf); // Wait for
        synchronization
        (LEDS_PORT->ODR)^=ORANGE_LED_BIT; // Toggle LED
    }

    return 0;
}
```

L'únic que té és un bucle infinit en el que intenta agafar un semàfor i, a continuació, canvia l'estat del led taronja fent servir la funció or exclusiva.

Veurem en detall el que passa durant l'execució:



En el moment en que es crea el fil fill, degut a que té mes prioritat, aquest comença a córrer deixant el procés del programa principal bloquejat. El fil fill pren el semàfor que és lliure des de que s'ha inicialitzat i canvia l'estat del led taronja. A continuació intenta tornar a agafar el semàfor, però no pot per que només es pot agafar una vegada. Queda, per tant, bloquejat.

En el moment que es bloqueja el procés fill, el programa principal continua corrent i arriba al seu bucle on crida **SLEEP\_MS**. Això fa que s'autobloquegi durant 300ms. Ara tenim els dos processos bloquejats i, per tant, pren el control de la CPU el procés de menys prioritat que és el **Idle Process**.

Quan passen 300ms, el programa principal es desbloqueja i deixa lliure el semàfor. Tan bon punt es deixa lliure el semàfor, pren el control el procés fill que és ara el més prioritari i està desbloquejat. Pren el semàfor i canvia l'estat del led. Però, només pot fer-ho una vegada perquè el semàfor queda ocupat tan bon punt l'agafa.

La seqüència es repeteix donant lloc a pampallugues al led verd amb un període de 600ms.

El que hem aconseguit és sincronitzar el funcionament dels dos processos que corren al mateix temps.

- Feu que es cridi a la funció **semaphoreExample** al principi de **main()** i comproveu que el funcionament del programa es l'esperat.

Com a demostració d'ús de semàfors demanem modificar el programa anterior. Afegirem un nou procés **thSem2** des del programa principal. Ara tenim tres processos: programa principal, **thSem** i **thSem2**.

Afegiu al programa principal la generació del nou procés **thSem2** amb la mateixa prioritat que **thSem**. Feu aquest nou procés **thSem2** igual que el procés **thSem** però amb el led **blau** i controlat per un nou semàfor **thSem** que també hem d'inicialitzar al programa principal.

Feu ara que el semàfor **semaf2** es desbloquegi, no al programa principal, si no cada 2 vegades que canviï el led a **thSem**. D'aquesta manera el procés **thSem** serà sincronitzat pel programa principal i el procés **thSem2** serà sincronitzat per **thSem**.

EP1 Feu una primera versió del codi que es demana.  
Descriuviu el que passarà quan es posi en marxa el programa.

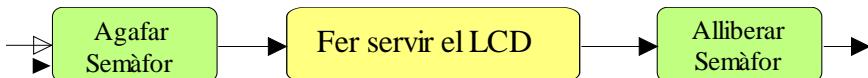
Els semàfors binaris, en realitat, només son un cas particulars dels semàfors amb comptador, (*Counting Semaphores*) aquests semàfors es poden agafar varies vegades abans de bloquejar el procés que l'intenta agafar. Tota la informació sobre aquest tipus de semàfors es troba a:

[http://chibios.sourceforge.net/html/group\\_\\_semaphores.html](http://chibios.sourceforge.net/html/group__semaphores.html)

### **R2.3Control d'elements comuns mitjançant Mutexes**

Els semàfors es poden fer servir, a part de per l'aplicació que hem vist de sincronització de processos, per controlar l'accés a un element comú com pot ser el cas del'LCD de la placa.

Podem assignar un semàfor binari al'LCD i fer que qualsevol procés, quan vulgui fer servir l'LCD hagi d'agafar el semàfor amb **chBSemWait**. Quan el procés acabi de fer servir l'LCD, l'allibera amb **chBSemSignal**. D'aquesta manera només un procés pot fer servir el LCD ja que el semàfor només es pot agafar una vegada abans d'alliberar-lo.



Si quan el procés vol fer servir l'LCD el semàfor està agafat per un altre procés, s'haurà d'esperar dins de la funció **chBSemWait** d'agafar semàfor fins que aquest quedí lliure.

Aquesta manera de controlar l'accés a elements comuns mitjançant semàfors pot tenir conseqüències fatales quan tenim processos amb diferents prioritats que fan servir més d'un recurs compartit. En concret es poden produir **deadlocks**, quan dos processos estan esperant cadascun a que s'alliberi un recurs que ha agafat l'altre, i problemes **d'inversió de prioritat**, quan un procés de mitja prioritat agafa la CPU per sobre d'un procés d'alta prioritat degut a que aquest últim està esperant un recurs que té un procés de baixa prioritat.

Es per això que per controlar l'accés a recursos és millor fer servir **Mutexes** que Semàfors. Mutex vol dir *Mutual Exclusion*. Aquestes eines, per tant, són específiques per controlar l'accés a un recurs per part d'un únic procés.

Els semàfors son variables i, com a tals, no pertanyen a cap procés. A ChibiOS/RT un **mutex** ve a ser com un tipus especial de semàfor binari que pertany al procés que l'agafa. Un **mutex** només pot ser alliberat pel procés que l'ha agafat, per tant, no podríem fer servir **mutexes** en el exemple del apartat R2.2.

Els **mutex** a ChibiOS/RT implementen un mecanisme que s'anomena heretament de prioritat. Posem que un procés A agafa un **mutex**. Posteriorment, un procés B més prioritari intenta agafar el mateix **mutex** però queda bloquejat per que el té el procés A. A partir d'aquest moment, el procés A treballarà a la prioritat del procés B per poder deixar el **mutex** tan ràpid com sigui possible i el pugui fer servir el procés B que es més prioritari. Aquest mecanisme elimina la possibilitat de que es doni una **inversió de prioritat** per part d'un tercer procés amb prioritat intermèdia entre els processos A i B.

Els **mutexes** es defineixen mitjançant una variable de tipus **Mutex**.

Un exemple de definició d'un **mutex** seria:

```
Mutex mutex_1;
```

On es definiria un **mutex** de nom **mutex\_1**

Les funcions principals per treballar amb **mutexes** son:

```
void chMtxInit ( Mutex * pmtx )
```

Inicialitza un **mutex** en l'estat lliure.

```
void chMtxLock (Mutex * pmtx )
```

Intenta agafar un **mutex**. Si està lliure, el **mutex** s'assigna al procés actual i aquest continua. Si no, el procés quedarà bloquejat fins que el **mutex** quedí lliure, llavors l'agafarà i continuarà.

```
Mutex * chMtxUnlock ( void )
```

Deixa lliure l'últim *mutex* que ha agafat el procés. Retorna un punter al *mutex* alliberat.

Observeu que els *mutex* s'han d'alliberar en ordre invers del que s'han agafat ja que sempre s'allibera l'últim *mutex* agafat. Això té també com a conseqüència que un procés no pot alliberar un *mutex* que no ha agafat.

La descripció completa dels Mutexes a ChibiOS/RT la podeu trobar a:

[http://chibios.sourceforge.net/html/group\\_\\_mutexes.html](http://chibios.sourceforge.net/html/group__mutexes.html)

## **R2.4Exemple d'ús de Mutexes**

Fixeu-vos en la segona part del fitxer **mutexes.c** on es troba la funció **mutexExample**

```
// Test where two threads access to the LCD
// The main thread write digits 0..9 on the first LCD line
// The child thread write letters A..Z on the second LCD line
void mutexExample(void)
{
    int32_t x,i,car;

    // Global initialization
    baseInit();

    // Initialize the LCD
    LCD_Init();

    // Clear the LCD and turn off the backlight
    LCD_ClearDisplay();
    LCD_Backlight(0);

    // Creates a child thread
    chThdCreateStatic(waThEM, sizeof(waThEM), NORMALPRIO, thMtx, NULL);

    // First digit to show
    car='0';

    // Infinite loop
    while (1)
    {
        for(i=0;i<20;i++)           // 20 times for each digit
            for(x=0;x<LCD_COLUMNS;x++)
                // For each column on the LCD...
                {
                    LCD_GotoXY(x,0);      // Jump to that column
                    LCD_SendChar(car);    // Write the digit
                    DELAY_US(17000);       // Some delay
                }
        if (++car>'9') car='0';     // Go to next digit
    }
}
```

Aquest programa inicia el sistema i l'LCD, crea un fil fill i entra en un bucle que escriu xifres a la primera línia del LCD.

El fil fill, definit per la funció **thMtx** també escriu al'LCD, però en aquest cas lletres a la segona filera.

```
// Child thread entry point
// Write letters on the second row of the LCD
static msg_t thMtx(void *arg)
{
    int32_t x,i,car;

    // First char to show
    car='A';

    // Infinite loop
    while (1)
    {
        for(i=0;i<20;i++)           // 20 times for each char
            for(x=0;x<LCD_COLUMNS;x++)
                // For each LCD column
                {
                    LCD_GotoXY(x,1);      // Jump to that column
                    LCD_SendChar(car);    // Write the char
                    DELAY_US2(10000);     // Some delay
                }
        if (++car>'Z') car='A';       // Go to next char
    }
    return 0;
}
```

Si posem en marxa la funció **mutexExample**, tots dos processos: programa principal i funció **thMtx** intentaran accedir al'LCD i, en algun moment, barrejaran les seves comunicacions donant lloc a *glitches* que podem veure sobre el LCD. Si li donem prou temps la pantalla del LCD fallarà completament.

- ⓘ Observareu que es fa servir la funció **DELAY\_US** en lloc de **SLEEP\_MS** per les pauses entre escriptures dels processos. Això es fa per que els dos processos treballin de manera asíncrona i els retards siguin acumulatius. Si féssim servir **SLEEP\_MS** tots dos processos quedarien sincronitzats i no es manifestarien errors en aquest programa. Això no vol dir que puguem treballar sense *mutexes*, si fem servir **SLEEP** en lloc de **DELAY**, l'únic que vol dir es que ens caldria un exemple més complex i ens costaria més trobar els errors. Penseu que els pitjors errors no són els evidents ja que aquests sempre s'eliminen a la fase de depuració, sinó els molt poc freqüents que són capaços d'arribar al producte final sense detectar.

Observareu tambè que al fil fill es fa servir la funció **DELAY\_US2** en lloc de **DELAY\_US**. D'aquesta manera fem servir un temporitzador independent per cada fil i no hi ha colisions entre ells.

- ⓘ Feu que es cridi a la funció **mutexExample** al principi de **main()** i comproveu com es produeixen errades a l'escriptura del LCD.

El LCD, en ser un element comú, ha de tenir arbitrat el seu accés. Ho farem amb ***Mutexes***.

Es demana modificar el programa de manera que tots dos processos hagin d'agafar un *mutex* abans de fer servir el LCD i l'alliberin després. Incloeu a **mutexes.c** la definició del *mutex* i no oblideu la seva inicialització dins de **mutexExample** abans de fer-lo servir.

**EP2** Feu una primera versió de la modificació del codi que demanem.

- Implementeu les modificacions demandades i comproveu que l'LCD opera lliure d'errors.

# Annexes

---

## A1 - Cable de Comunicacions EXT1

### Introducció

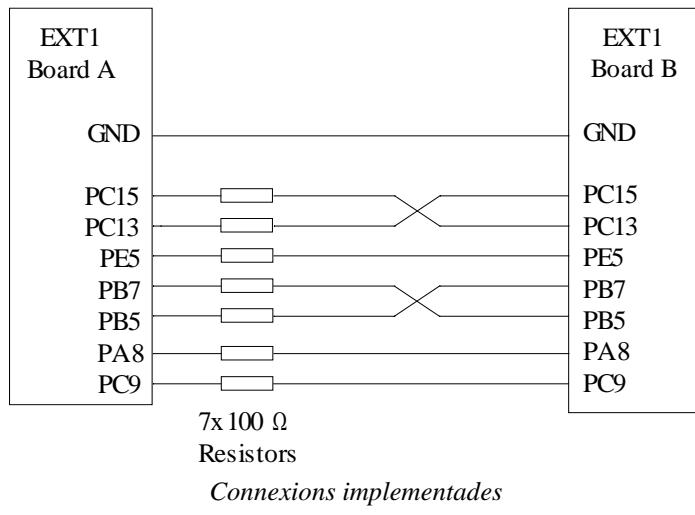
Dins les assignatures SEBM i SBM es possible que alguns grups, després d'acabar les pràctiques obligatòries, tinguin temps de desenvolupar continguts opcionals. Alguns continguts opcionals estan reglats mitjançant pràctiques o apartats opcionals. Altres continguts opcionals són més lliures i depenen de les propostes que facin els grups de laboratori. Es dins d'aquest context on s'emmarca aquest cable de comunicacions.



*Cable de Comunicacions EXT1*

El cable de comunicacions EXT1 permet comunicar el port extern EXT1 de la placa de laboratori amb el port EXT1 de l'altre placa. Això permet implementar mecanismes de comunicació entre plaques.

El cable implementa les següents connexions entre dues plaques de pràctiques.

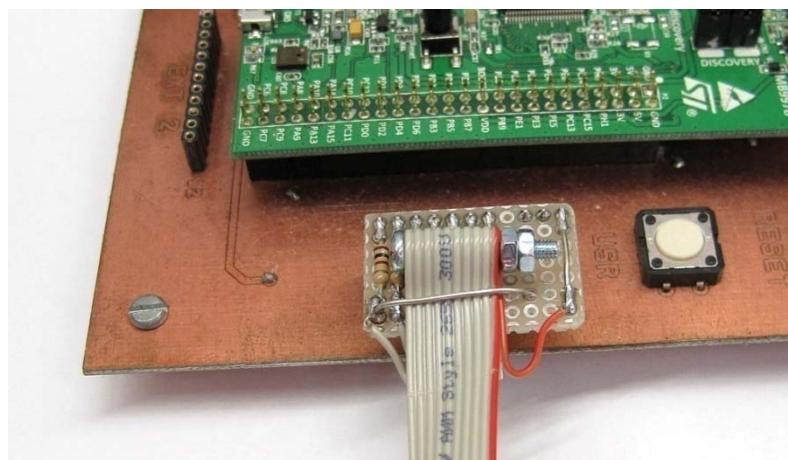


Les resistències de  $100\Omega$  que hi ha a cadascuna de les línies, excepte GND, permeten protegir les plaques de pràctiques en el cas de que, per error, s'intenti forçar dos valors lògics diferents en els dos extrems d'una de les línies.

Les connexions de les línies GPIO PE5, PA8 y PC9 són directes entre les dues plaques.

Les connexions PC15 y PC13 es troben creuades entre les dues plaques de la mateixa manera que ho estan les connexions PB7 y PB5. Això permet implementar comunicacions fent servir la mateixa configuració de pins de transmissió y recepció.

Per fer servir el cable només cal connectar cada extrem del cable a una placa de pràctiques tal y como es mostra a la figura de sota. El cable, correctament connectat, sempre ha de sortir cal a l'exterior de la placa.



*Cable sortint d'una placa de pràctiques*

## Exemples de protocols de comunicació

A continuació es mostren alguns exemples de mecanismes de comunicació entre dues plaques que es poden implementar. Aquesta llista d'opcions no inclou totes les possibilitats d'ús del cable per establir una comunicació donat que sempre es poden definir altres mecanismes no inclosos en aquest document. Per exemple únicament es descriuran comunicacions bidireccionals. Addicionalment, a cada exemple es fa una proposta d'assignació de línies GPIO a les diferents senyals que pot ser modificada.

La següent taula mostra les línies GPIO disponibles a EXT1 i si el cable implementat fa o no creuament d'un parell de GPIOs.

Pin	Funció	Tipus	Funcions alternatives
5	PC15	Creuat	
6	PC13		
7	PE5	Directe	TIM9-CH1
8	PB7	Creuat	TIM4-CH2 / <b>USART1-RX</b>
9	PB5		TIM3-CH2
10	PA8	Directe	MCO1 / TIM1-CH1 / <b>I2C3-SCL</b>
11	PC9	Directe	MCO2 / TIM3-CH4 / TIM8-CH4 / <b>I2C3-SDA</b>

*Taula de GPIO a EXT1*

Observant el senyals disponibles al connector EXT1 podem veure que hi han associades algunes línies de perifèrics hardware del microcontrolador. Per exemple, podríem fer servir PA8 i PC9 per establir una connexió sèrie síncrona Half-Duplex fent servir el protocol I2C o podríem fer servir PB7 per generar un senyal asíncron sèrie fent servir el perifèric USART. Tot i que podem fer servir aquests perifèrics, l'objectiu d'establir la comunicació entre dues plaques no es tant saber fer servir els perifèrics hardware del microcontrolador sinó entendre el protocol de comunicació. Es per això que no hi ha problema en fer servir control a nivell de GPIO (*bit banging*) per totes les funcions de comunicació. En tot cas, per raons pràctiques, es recomana fer servir interrupcions, especialment en les rutines de recepció.

Per simplicitat, no es mostraran a les connexions ni la línia de massa (GND) ni les resistències de protecció.

## Comunicació Sèrie Síncrona *Full Duplex*

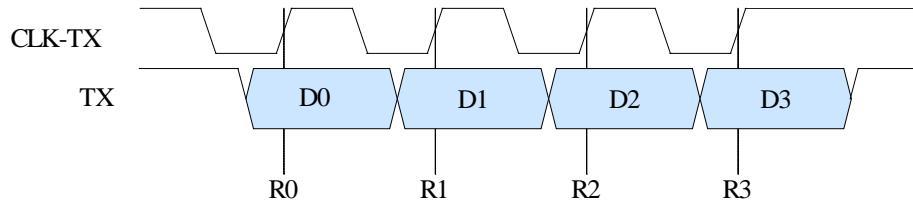
Aquest tipus de connexió es pot implementar definint per cada placa la següent assignació de les línies disponibles a EXT1.

GPIO	Type	Usage
PC15	Cross	TX
PC13		RX
PB7	Cross	CLK-TX
PB5		CLK-RX

*Proposta d'assignació per Serial Síncrona Full Duplex*

Aquest tipus de comunicació fa servir dos canals independents per comunicar dues plaques A i B. Un va de A a B i l'altre va de B a A.

Cada placa genera un senyal TX en PC15 associat a un rellotge de transmissió CLK-TX a PB7. Per tant, PC15 y PB7 se han de configurar en mode sortida *push-pull*. Per transmetre una dada de 4 bits podria fer-se servir un mètode com l'indicat a continuació:



*Exemple de transmissió síncrona de 4 bits*

Como es pot veure, cada dada s'actualitza al mateix temps (o just després) que baixa el senyal de rellotge. En aquest exemple hem definit un valor alt "1" per defecte en estat de repòs (*idle*).

El receptor, llegirà un bit en RX (PC13) a cada flanc de pujada de CLK-RX (PB5). Fent servir interrupcions garantirem la sincronització correcta de les lectures al flanc de pujada de CLK. Les línies GPIO PC13 y PB5 hauran de ser configurades en mode entrada.

## Sincronització de trama

Fent servir comunicacions síncrones es possible transmetre dades de qualsevol longitud. Amb objecte de sincronitzar les dades s'ha de definir la estructura lògica de la trama de dades enviada y la manera de sincronitzar les dades quan la recepció es desincronitza momentàniament.

Podríem afegir dues línies addicionals per fer la sincronització de trama però això augmentaria els nostres requeriments de hardware. A més, no disposem de més parells creuats al cable.

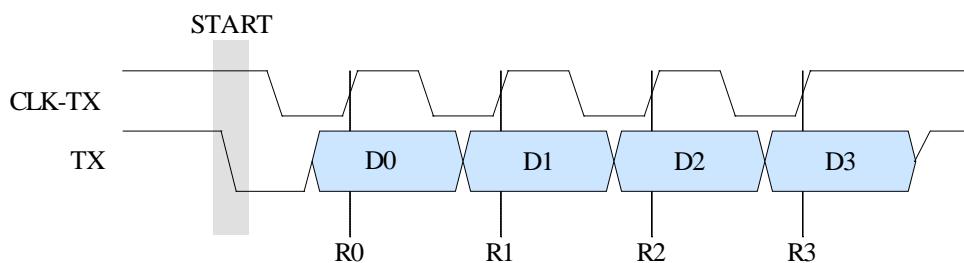
A nivell pràctic hi ha dues maneres de fer la sincronització de trama. La primera es incloure-la a les propies dades. Per exemple, en transmissions de 4 bits podríem enviar un "1" després de cada quatre bits. Això vol dir que mai podríem tenir 5 "0"s setguits ja que hi-ha un "1" entre cada 4 bits.

Per fer la sincronització de trama podríem enviar una seqüència especial de 5 "0"s. La següent figura mostra aquest tipus de sincronització on transmetem un conjunt de dades A...Z de quatre bits cadascuna i sincronitzem la trama cada vegada que tornem a començar des de la "A".

**0 0 0 0 0 A0 A1 A2 A3 1 B0 B1 B2 B3 1 .... 1 Z0 Z1 Z2 Z3 0 0 0 0 0 A0 A1 A2 A3 ....**

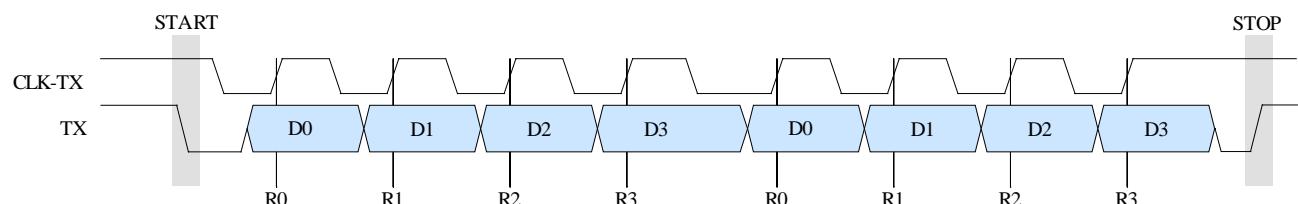
*Exemple de sincronització de trama per dades de 4 bits*

El segon mètode de fer la sincronització de trama es fer servir condicions especials en els flancs dels senyals de dades i rellotge. Si observem el cronograma del exemple de transmissió veurem que les dades sempre canvien quan el rellotge es baixa. Donat que mai no es dona la condició de que les dades canvien amb rellotge alt, podem fer servir aquest cas especial per sincronitzar la trama.



*Exemple de sincronització de trama amb un flanc de baixada a TX*

Si el nombre de dades enviar a cada trama té la mateixa mida, basta tenir aquesta condició de sincronització. Però, si la trama pot tenir mida variable pot ser interessant definir un condició de acabament de la trama complementaria a la d'arrencada.



*Exemple de sincronització de trama de dues dades de 4 bits amb condició d'inici i de finalització de trama*

Aquest tipus de sincronització de trama fent servir la línia de dades TX es fa precisament d'aquesta manera al busos sèrie I2C com el que tenim a la placa per comunicar els controls del DAC d'audio amb el microcontrolador.

Tot i que la sincronització de trama la hem considerat per aquest cas síncron full-duplex, en general es poden fer servir mètodes similars per tots els tipus de comunicaciones síncrones.

## Comunicació Sèrie Síncrona *Full Duplex* amb *Master*

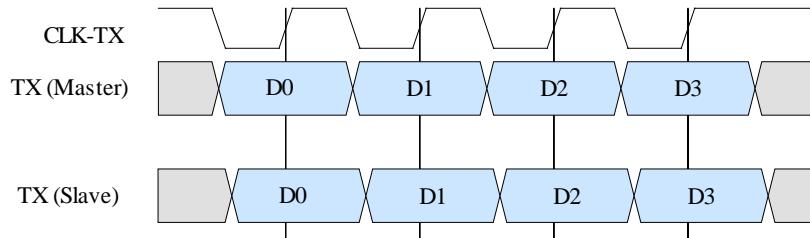
En el cas anterior, les dues plaques poden iniciar la comunicació amb la seva línia TX de manera independent. Una comunicació te un *master* es defineix com aquella en la que una placa controla el flux de la comunicació. L'altra placa, que no controla la comunicació, es denomina *slave*. El bus **SPI** es un exemple de comunicació *Full Duplex Síncrona*.

Podem establir una comunicació síncrona fent servir, per exemple, la següent distribució de senyals:

GPIO	Type	Usage
PC15	Cross	TX
PC13		RX
PB7	Cross	CLK-TX (Master only)
PB5		CLK-RX (Slave only)

*Proposta d'assignació per Serial Síncrona Full Duplex amb Màster*

Les connexions son molt similars al caso de anterior. La diferencia fonamental es que el senyal CLK-TX només s'implementa a la placa *master* y el senyal CLK-RX només s'implementa a la placa *slave*.



*Senyals en el mode Sèrie Síncron Full Duplex amb Màster*

Per aquest mode de funcionament la placa *master* actua igual que el caso síncron sense *master* anterior. En el cas de la placa *slave*, aquesta ha de llegir l'estat del senyal CLK-RX i ha d'escriure el seu bit TX en detectar cada flanc de baixada del rellotge. Totes dues plaques llegiran en el flanc de pujada de clk. En el cas de la placa *master*, donat que genera el rellotge, només cal que llegeixi la línia RX després de generar el flanc de pujada. En el caso de la placa *slave*, se ha de llegir la línia de rellotge, preferentment fent servir interrupcions, per detectar la presencia del flanc de pujada.

A mes de caldre una línia menys, la comunicació amb *master* permet simplificar en alguns cassos el codi de la aplicació ja que es sincronitza la emissió y recepció. Observeu que en el caso sense *master* tots dos canals de comunicació eren completament independents.

## Comunicació Sèrie Síncrona *Full Duplex* amb dos *Master*

En el cas de tenir una comunicació síncrona amb *master*, no es necessari que una placa faci sempre de *master*. Es possible definir un protocol en el qualsevol de les dues plaques pugui iniciar la transacció actuant como a *master*.

Podem definir un protocol fent servir tres línies, igual que al cas anterior. Farem servir, però, una línia sense creuament como PE5 pel rellotge.

GPIO	Type	Usage
PC15	Cross	TX
PC13		RX
PE5	Direct	CLK

*Proposta d'assignació per Serial Síncrona Full Duplex Multimaster*

Donat que les dues plaques poden iniciar la comunicació como a *master*, convé que no puguin forçar valors diferents a la línia CLK comuna. Per això podem definir PE5 a totes dues plaques com a sortida *Open Drain* amb *Pull-Up*. En estat de repòs totes dues plaques deixen CLK en nivell alto i llegeixen l'estat de la línia. Quan una placa desitja iniciar una comunicació como a *master*, posa CLK a “0”. L'altre placa, en detectar el flanc de baixada sap que ha de actuar como a *slave*.

Existeix la possibilitat, tot i que poc probable, de que les dues plaques intentin actuar como a *master* al mateix temps. Això donaria lloc a una col·lisió. Una manera de detectar aquesta col·lisió es que totes dues plaques llegeixin l'estat de la línia CLK quan fan de *master*. Si detecten que CLK es troba a “0” quan hauria d'estar a “1” vol dir que l'altre placa està intentant controlar CLK i, per tant, actua també como a *master*.

## Comunicació Sèrie Síncrona *Half Duplex*

Totes les comunicacions anteriors eren de tipus *Full Duplex*. Es a dir, totes dues plaques podien transmetre dades a l'altre al mateix temps. Una comunicació *Half Duplex* únicament permet comunicar en un sentit al mateix temps. Això permet reduir el nombre de línies requerit a numès dos. Un exemple d'aquest tipus de comunicació es l' implementat a l'estàndard **I2C**.

Podem implementar una comunicació síncrona *Half Duplex*, per exemple fent servir la distribució de línies indicades a continuació:

GPIO	Type	Usage
PA8	Direct	TX-RX
PC9	Direct	CLK

*Proposta d'assignació per Serial Síncrona Half Duplex*

Farem servir PA8 per les dades i PC9 pel rellotge. Totes dues sense creuar. Per tal d'evitar col·lisions perilloses a nivell de hardware configurarem tant PA8 como PC9 com a sortides *Open Drain* con *Pull-Up*. El protocol pot ser similar al cas anterior Síncron *Full Duplex* amb dos *Master*. La diferència es que la placa que actua como a *master* controlarà CLK a PC9 y farà servir PA8 per transmetre i la placa que actua como a *slave* farà servir PC9 i PA8 para rebre el rellotge i les dades respectivament.

Una vegada més s'ha de definir un protocol per evitar col·lisions si les dues plaques intenten transmetre al mateix temps. Es possible implementar tot el protocol I2C per aquesta comunicació, ja que aquest inclou la detecció de col·lisions. Tot i això, la complexitat associada no es realment necessària donat que en el nostre cas només comuniquem dues plaques mentre que l'estàndard I2C permet comunicar múltiples components.

Una variant d'aquest tipus de comunicació consistiria en que una de les dues plaques actuï sempre como a *master* y controli el rellotge CLK. Dintre de la trama de dades enviada es donaria la indicació a l'altre placa de quan ha de posar informació a PA9.

## Comunicació Sèrie Asíncrona *Full Duplex*

Totes les comunicacions descrites anteriorment son de tipus síncron. Fer servir comunicacions síncrones simplifica molt l' implementació donat que no cal fer servir requisits temporals molt estrictes.

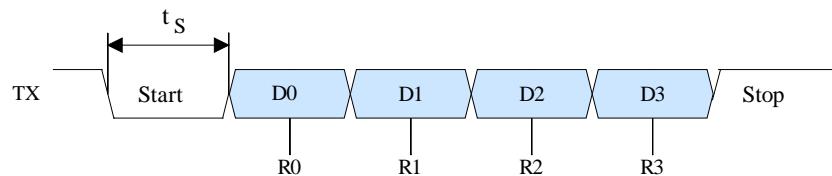
A les comunicacions asíncrònies ens estalviem l'enviament del rellotge fent servir una base de temps comuna entre les dues plaques.

Una connexió asíncrona *Full Duplex* es pot implementar definint per cada placa la següent assignació de les línies disponibles a EXT1.

GPIO	Type	Usage
PC15	Cross	TX
PC13		RX

*Proposta d'assignació per Serial Asíncrona Full Duplex*

Aquest tipus de comunicació es similar al cas síncron *Full Duplex* sense *master* però, en ser asíncrona, no es fa servir cap senyal de rellotge.



*Transmissió asíncrona de 4 bits*

A la comunicació síncrona, la longitud de cada bit enviat podia ser diferent donat que cada dada es sincronitzava amb el rellotge. En el cas asíncron, donat que no s'envia el rellotge, cada símbol (1 bit en el nostre cas) ha de tenir una longitud constant que anomenarem **temps de símbol**  $t_s$ .

Suposant que la línia en repòs es troba a nivell alt, podem senyalitzar l'inici de la comunicació posant la línia a nivell baix durant un temps  $t_s$ . Després podem transmetre cada bit durant un temps  $t_s$ . Al final deixarem la línia en repòs al menys un temps  $t_s$  base d'enviar un altre dada.

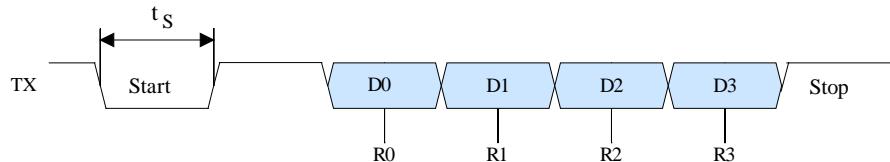
Degut a que el temps de símbol ha de ser conegut i sempre el mateix, s'ha de vigilar la seqüènciació de la transmissió. Una manera de fer-ho es fent servir un temporitzador hardware.

El receptor restarà a l'espera a detectar la baixada de RX que indica l'inici de la transmissió d'una dada. A partir d'aquest instant esperarà un temps  $1,5t_s$  abans de llegir el primer bit y, es d'aquest punt, es llegirà un bit cada vegada que passi un temps  $t_s$ .

A la comunicació asíncrona tant l'emissor como el receptor han d'estar d'acord en el temps  $t_s$  i en el nombre de símbols de cada dada enviada.

Degut a que es impossible garantir valors de  $t_S$  exactament iguals a l'emissor i al receptor, l'instant de captura a RX s'anirà desviant poc a poc del centre de cada símbol enviat. Aquest fet limita la longitud màxima que pot tenir una dada.

Es possible definir una comunicació asíncrona a la que el temps de símbol es calculi automàticament. Una manera de fer-ho es afegint un bit alt després del bit de *start* de manera que puguem deduir  $t_S$  a partir de l'amplada del bit de *start* tal y como es mostra en a la següent figura.



*Transmissió asíncrona de 4 bits que permet deduir  $t_S$*

A les comunicacions asíncrònies no cal fer servir condicions especials de sincronització de trama ja que la comunicació es sincronitza dada a dada. Si un conjunt de dades constitueix una trama lògica, s'ha de definir la sincronització fent servir les propies dades.

## Comunicación Sèrie Asíncrona *Half Duplex*

Aquest es el últim cas de comunicació que descriurem. Es minimalist per que fa servir una única línia, llevat de massa (GND).

Aquest tipus de comunicació, tot i que amb una implementació diferent, es la que es fa servir al estàndard **1-Wire**.

Podem establir aquesta comunicació amb qualsevol línia no creuada. Per exemple:

GPIO	Type	Usage
PE5	Direct	TX/RX

*Proposta d'assignació per Serial Asíncrona Half Duplex*

Igual que altres casos de línies compartides configurarem la línia de comunicació PE5 com a sortida *Open Drain* amb *Pull-Up*. El principi d'operació es similar al cas anterior síncron *Half Duplex*. La diferència es que en aquest cas, la placa que inicia la transacció i actua com a *master*, farà servir una comunicació asíncrona transmetent un bit de *start* seguit dels bits de dades i acabant amb, al menys, un bit de *stop*.

## Altres Opcions

Totes les opcions descrites no esgoten ni de lluny totes les possibles comunicacions que es poden establir amb el Cable de Comunicacions EXT1. Algunes opcions addicionals que es poden provar son:

- Comunicació Asíncrona *Full Duplex* de 2 bits simultanis.
- Comunicació Asíncrona de *Half Duplex* de 7 bits simultanis.
- Comunicació Síncrona de *Half Duplex* de 6 bits simultanis.

Tots els cassos considerats fins ara eren Síncrons (Enviant el rellotge amb un senyal separat) o Asíncrons (Sense enviar el rellotge). Existeix un altre categoria de comunicacions a les que s'envia el rellotge codificat dins de les dades. En aquests casos tenim a la vegada els avantatges de la velocitat síncrona i el reduït nombre de línies asíncron. Aquestes avantatges tenen però el preu d'una major complexitat, especialment a la descodificació.

Alguns exemples amb rellotge codificat a les dades són:

- Comunicació Sèrie amb codificació Manchester y recuperació de rellotge
- Comunicació Sèrie amb codificació avançada como 8B/10B

## Verificació de la integritat

En tots els casos descrits existeix la possibilitat de corrupció de l' informació degut, per exemple, al soroll. A les dades enviades es possible afegir verificació o correcció de les dades fent servir bits de paritat, càlculs de redundància cíclica (CRC) o bits de correcció basats en distàncies mínimes.

## Connectors de test

Alguns dels tipus comunicacions que hem descrit son simmètriques. Es a dir, totes dues plaques implementen el mateix codi per establir la comunicació. Aquest es el cas en:

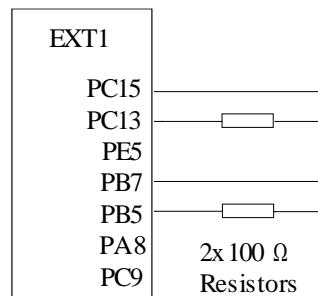
- Comunicació Sèrie Síncrona Full Duplex
- Comunicació Sèrie Asíncrona Full Duplex

Per simplificar el desenvolupament de comunicaciones simmètriques, a la capsula on hi ha el cable de test trobareu dos connectors de test com el que es mostra a la figura:



*Connector de test*

Aquest connector creua les línies PC15 amb PC13 i PB7 amb PB5. Igual que al cable descrit anteriorment, es fan servir resistències de  $100\Omega$  per protegir el MCU en cas de fer malament la configuració de les línies GPIO.



*Connexions al connecto de test*

Amb aquest connector podem provar les rutines de transmissió i recepció, en el cas de protocols simmètrics full duplex, sobre una única placa.