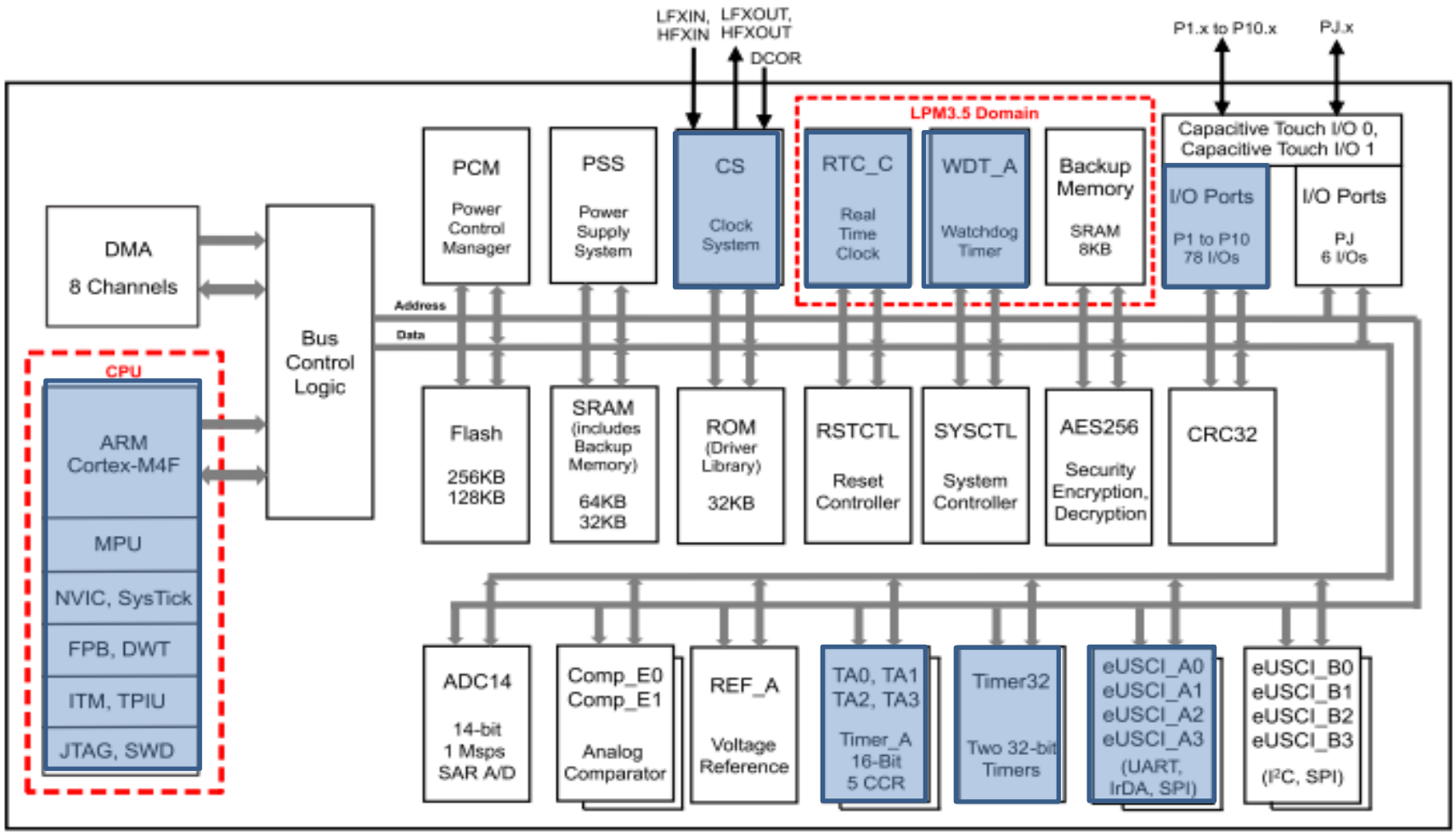
PROGRAMACIÓ D'ARQUITECTURES ENCASTADES

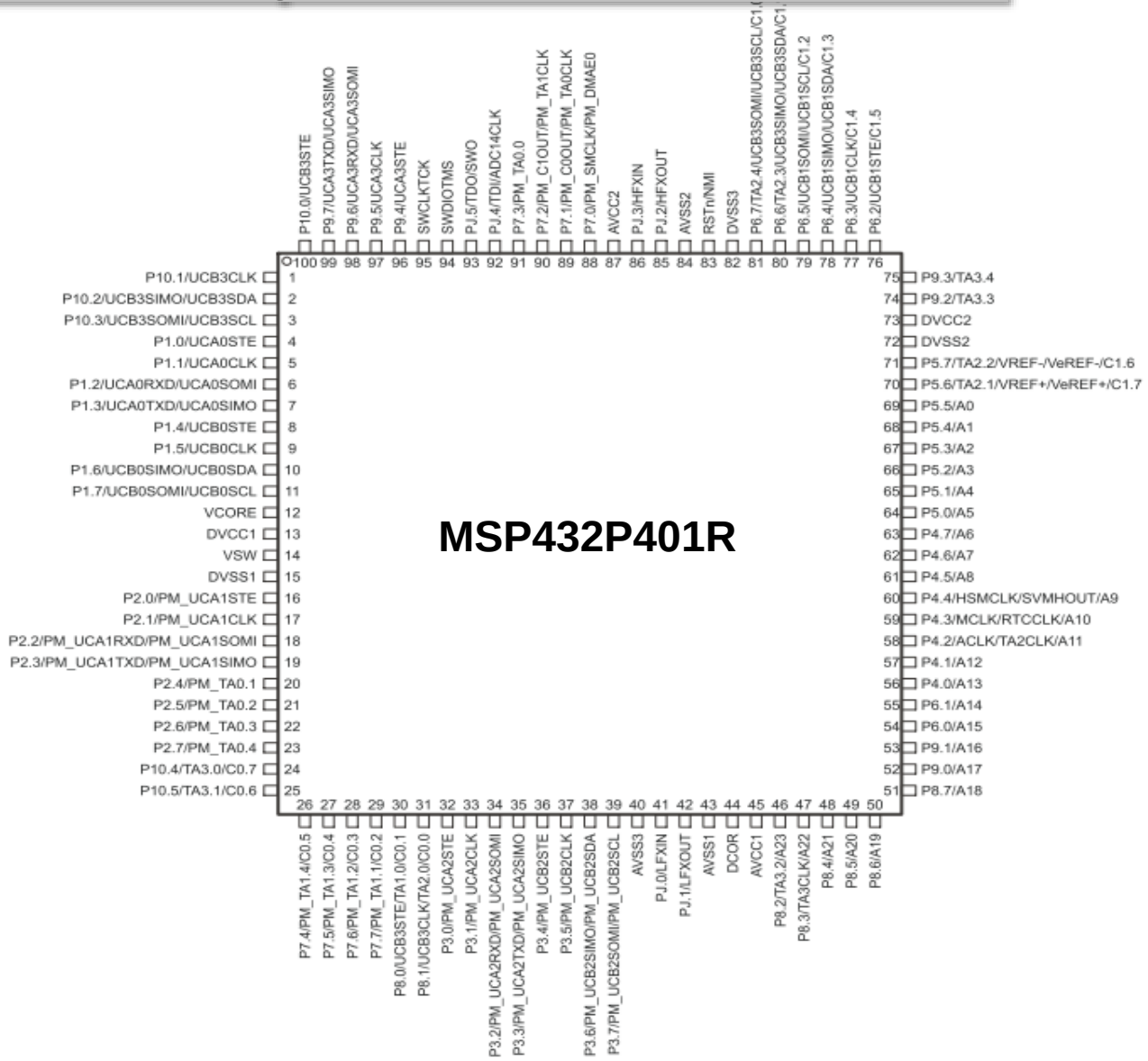
Resum

Diagrama de blocs Funcional del MSP432P401



Encara que pot semblar que els diferents blocs són completament independents entre ells, resulta que comparteixen pins i això introduirà limitacions.

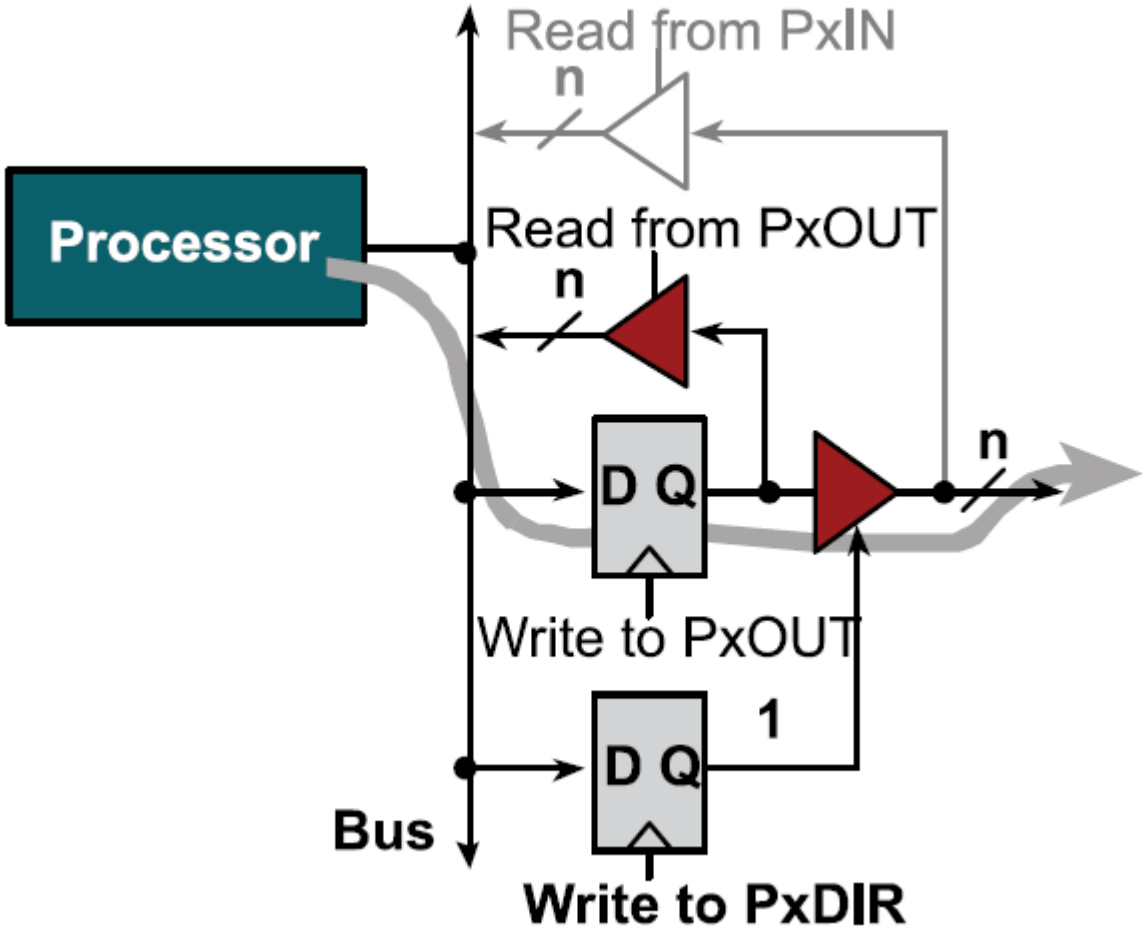
Assignació de pins del MSP432P401



E/S Digitals (GPIOs, *General Purpose Input Outputs*)

- Aquests pins s'agrupren en ***ports* de 8 pins**. Nosaltres els controlarem com aquests grups mitjançant variables o registres de mida un byte (en C *unsigned char* o *uint8_t*).
- Així, al MSP432P401 tenim el *port* P1 que agrupa a 8 *pins*, els quals denominarem **P1.0, P1.1, P1.2... P1.7**. El "problema" és que no podem accedir directament amb les nostres instruccions de programa a un dels pins, si no que sempre haurem de fer referència a tot el port. Això implica que hem d'anar en compte a l'hora de modificar l'estat d'un o diversos pins sense modificar la resta de pins del *port*. Més endavant veurem com fer-ho de forma adequada.
- En un instant determinat **només pot exercir una** de les funcions assignades.
- Existeixen una sèrie de **registres de configuració** per controlar quina de les **funcions assignades** és la que volem que faci **en cada moment**.
- En qualsevol moment del programa podem **re-programar** el funcionament de cada pin.
- La **configuració** es fa **a nivell de port** (grup de 8 pins) però **cada pin** del port es pot configurar amb la seva funció **independentmen**t dels altres.

Diagrama de blocs d'un GPIO



Configuració de Ports Digitals

MSP432P4xx *Technical reference manual: Pags. 482-484*

10.2 Operació dels *Ports* Digitals *(GPIOs)*

- **Registres d'Entrada PxIN**: Cada bit del registre reflecteix el valor del senyal d'entrada del pin I/O corresponent, quan el pin està configurat com a una entrada digital.
- **Registres de Sortida PxOUT**: Cada bit del registre és el valor que volem que surti pel pin I/O corresponent, quan el pin està configurat com a una sortida digital.
- **Registres de Selecció de Funció PxSEL0 i PxSEL1**: Quan un pin està multiplexat amb funcions d'altres perifèrics, aquests registres permeten triar quina de les funcions volem:

PxSEL1	PxSEL0	Configuració I/O
0	0	Funciona com a I/O digital (GPIO)
0	1	Primera funció alternativa
1	0	Segona funció alternativa
1	1	Tercera funció alternativa

Configuració de Ports Digitals

MSP432P4xx *Technical reference manual: Pags. 482-484*

10.2 Operació dels *Digital I/O*

- **Registres de Sentit PxDIR**: Quan amb els registres PXSELy seleccionem treballar com a GPIO, cada bit del registre PxDIR selecciona si volem que el pin I/O corresponent sigui d'Entrada o de Sortida.
  - ✓ Bit = 0: El pin del port el definim com Entrada.
  - ✓ Bit = 1: El pin del port el definim com Sortida.
- A més, si hem programat un *pin* com entrada (*input*), hem de tenir en compte els registres de les resistències *Pullup/Pulldown*: **PxREN**.

PxDIR	PxREN	PxOUT	Configuració I/O
0	0	x	Entrada sense resistència de *pullup/pulldown*
0	1	0	Entrada amb resistència de *pulldown*
0	1	1	Entrada amb resistència de *pullup*
1	x	X	Sortida

Manipulació dels registres

Ara volem escriure un 1 en P1.0 i deixar la resta igual, si fem **P1OUT=0x01**, estem modificant tots els pins de sortida (P1.0 a 1 i P1.1, P1.2 i P1.3 a 0, P1.4-P1.7 són entrades i no veuran modificats per un operació de sortida)
Per evitar aquesta situació hem de fer servir els operadors lògics bit a bit, **AND (&), OR (|), XOR (^) i Complement a 1 (~)**.

Al nostre cas hauríem d'haver fet: **P1OUT |= 0x01.**
OR: Posar a 1 els pins on escrivim 1

I si volem ficar un 0 al mateix pin, sense modificar la resta: **P1OUT &= ~(0x01).**
AND: Posar a 0 els pins on escrivim 0

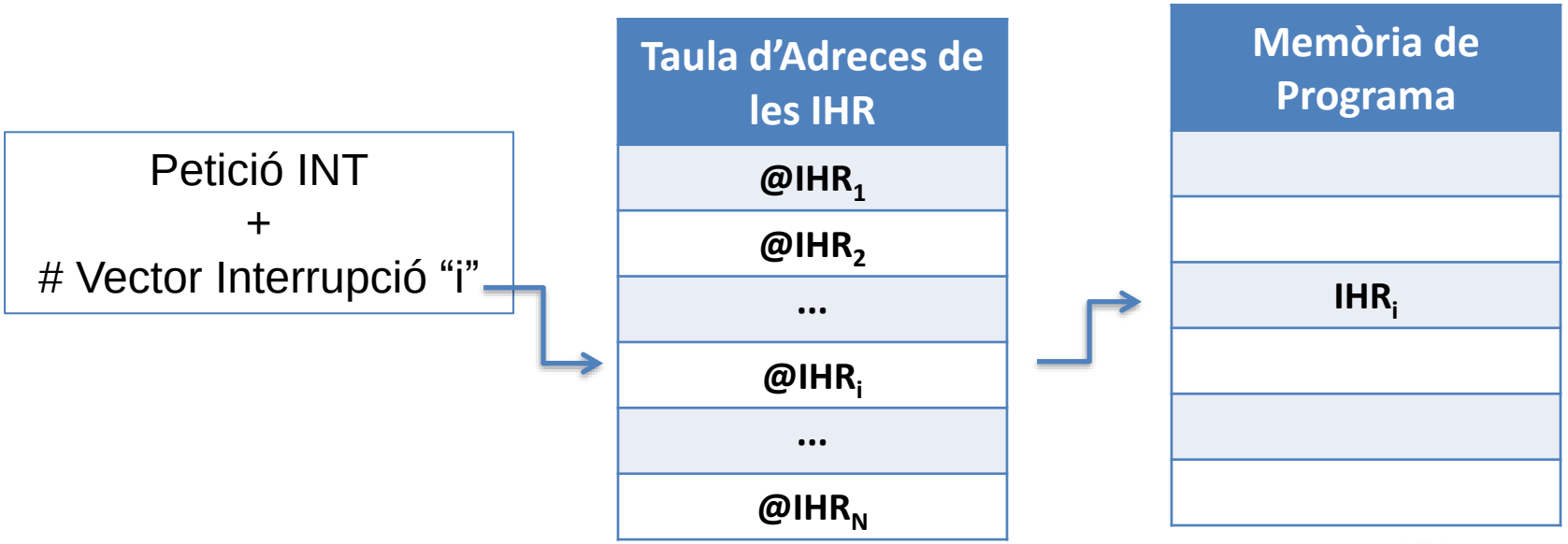
Si el que volem és invertir l'estat del mateix pin: **P1OUT ^= 0x01.**
XOR: Invertir el valor dels pins on escrivim 1.

Això és també vàlid per modificar bits de registres de configuració, sense modificar l'estat de la resta de bits del registre.
Exemple, volem configurar P1.0 de sortida sense modificar la resta:
P1DIR |= 0x01.

Gestió mitjançant un Controlador d'Interrupcions

Aquests sistemes es basen en que la identificació de la font d'interrupció, les prioritats i l'emmascarament, es fan per hardware.

- Cada dispositiu que pot generar interrupcions té associat un identificador, anomenat **Vector d'Interrupció**, que ha de proporcionar al processador quan demana una interrupció.
- Cada vector d'interrupció té asociada una posició a una taula on es guarden les adreces de les rutines d'atenció a les interrupcions (IHR_i).



Sistema d'Interrupcions del MSP432P401

Per fer servir les interrupcions de qualsevol recurs, hem de seguir una sèrie de passes:

- Habilitar les interrupcions. Al nostre processador es fa a 3 nivells:
  1. **A nivell de dispositiu** (s'ha de consultar el *Technical reference* per a cada dispositiu).
  2. **A nivell del controlador d'interrupcions del processador**, anomenat **NVIC** (*Nested Vectored Interrupt Controller*).
  3. **A nivell global** al registre de "status" del processador, fent servir la instrucció: **__enable_interrupt();**
- De fet, al primer pas que hem indicat, a nivell de dispositiu, podem emmascarar (inhabilitar) algunes interrupcions i mantenir altres actives.
- El controlador d'interrupcions també gestiona un sistema de prioritats, per aquelles situacions en que més d'una interrupció s'activi simultàniament.

Exemple d'Interrupcions a un *port* GPIO

Suposem que volem fer servir tot el *port* 2 com a GPIOs d'entrada i habilitar tots els seus pins per generar interrupcions:

A nivell de dispositiu (*Technical reference*, cap. 10.4, pàg. 520-521):

```
P2IE |=0xFF; //Habilitem a nivell de dispositiu les interrupcions als 8 pins (P2.0 a P2.7)
P2IES &=0x00; //Volem que les interrupcions saltin al flanc de pujada L->H
```

A nivell de NVIC:

```
NVIC->ICPR[1] |= BIT4; //Ens assegurem que no quedin interrupcions pendents al port 2
NVIC->ISER[1] |= BIT4; //Habilitem les interrupcions al port 2
```

A més haurem de posar en algun moment: **__enable_interrupt();**

Una bona pràctica és agrupar totes les habilitacions d'interrupcions a nivell de NVIC a una funció, per exemple "void init_interrupcions(void)....".
Les habilitacions a nivell de dispositiu és preferible fer-les a la inicialització de cada dispositiu.
I, sobretot, mai executar el "__enable_interrupt();" abans d'haver fet les dues coses anteriors.

NVIC *(Nested Vectored Interrupt Controller)*

Pot gestionar **fins a 64 vectors d'interrupció, amb 8 nivells de prioritat.**
Per gestionar el **NVIC** a nivell de programació, hi ha 7 tipus de registres, dels quals nosaltres en farem servir només 2:

1. **2 registres** de activació d'interrupcions, **ISER0 i ISER1** (*Interrupt Set Enable Register*). **Cada un d'ells de 32 bits**. Amb aquests 2 registres farem la **habilitació a nivell de NVIC** de cada font d'interrupció.
2. 2 registres d'esborrat d'interrupcions pendents, **ICPR0 i ICPR1** (*Interrupt Clear Pending Register*) també de 32 bits.

La resta de registres són per modificar prioritats, marcar interrupcions com a pendents, comprovar si una interrupció està activa, o activar-les per *software*. Però, al menys per ara, no els farem servir.

També necessitarem la informació que hi ha a la taula 6-39 del *Datasheet* (pàg.117 i 118) que ens especifica el significat de cada bit per fer servir els registres ISERx i ICPRx.

NVIC *(Nested Vectored Interrupt Controller)*

NVIC INTERRUPT INPUT	SOURCE	FLAGS IN SOURCE
INTISR[3]	WDT_A	
INTISR[4]	FPU_INT (2)	Combined interrupt from flags in the FPSCR (part of Cortex-M4 FPU)
INTISR[5]	FLCTL	Flash Controller interrupt flags
INTISR[6]	COMP_E0	Comparator_E0 interrupt flags
INTISR[7]	COMP_E1	Comparator_E1 interrupt flags
INTISR[8]	Timer_A0	TA0CCTL0.CCIFG
INTISR[9]	Timer_A0	TA0CCTLx.CCIFG (x = 1 through 4), TA0CTL.TAIFG
INTISR[10]	Timer_A1	TA1CCTL0.CCIFG
INTISR[11]	Timer_A1	TA1CCTLx.CCIFG (x = 1 through 4), TA1CTL.TAIFG
INTISR[12]	Timer_A2	TA2CCTL0.CCIFG
INTISR[13]	Timer_A2	TA2CCTLx.CCIFG (x = 1 through 4), TA2CTL.TAIFG
INTISR[14]	Timer_A3	TA3CCTL0.CCIFG
INTISR[15]	Timer_A3	TA3CCTLx.CCIFG (x = 1 through 4), TA3CTL.TAIFG
INTISR[16]	eUSCI_A0	UART or SPI mode TX, RX, and Status Flags
INTISR[17]	eUSCI_A1	UART or SPI mode TX, RX, and Status Flags
INTISR[18]	eUSCI_A2	UART or SPI mode TX, RX, and Status Flags
INTISR[19]	eUSCI_A3	UART or SPI mode TX, RX, and Status Flags
INTISR[20]	eUSCI_B0	SPI or I^2C mode TX, RX, and Status Flags (I^2C in multiple-slave mode)
INTISR[21]	eUSCI_B1	SPI or I^2C mode TX, RX, and Status Flags (I^2C in multiple-slave mode)
INTISR[22]	eUSCI_B2	SPI or I^2C mode TX, RX, and Status Flags (I^2C in multiple-slave mode)
INTISR[23]	eUSCI_B3	SPI or I^2C mode TX, RX, and Status Flags (I^2C in multiple-slave mode)
INTISR[24]	ADC14	IFG[0-31], LO/IN/HI-IFG, RDYIFG, OVIFG, TOVIFG
INTISR[25]	Timer32_INT1	Timer32 Interrupt for Timer1
INTISR[26]	Timer32_INT2	Timer32 Interrupt for Timer2
INTISR[27]	Timer32_INTC	Timer32 Combined Interrupt
INTISR[28]	AES256	AESRDYIFG
INTISR[29]	RTC_C	OFIFG, RDYIFG, TEVIFG, AIFG, RT0PSIFG, RT1PSIFG
INTISR[30]	DMA_ERR	DMA error interrupt
INTISR[31]	DMA_INT3	DMA completion interrupt3
INTISR[32]	DMA_INT2	DMA completion interrupt2
INTISR[33]	DMA_INT1	DMA completion interrupt1
INTISR[34]	DMA_INT0 (3)	DMA completion interrupt0
INTISR[35]	I/O Port P1	P1IFG.x (x = 0 through 7)
INTISR[36]	I/O Port P2	P2IFG.x (x = 0 through 7)
INTISR[37]	I/O Port P3	P3IFG.x (x = 0 through 7)
INTISR[38]	I/O Port P4	P4IFG.x (x = 0 through 7)
INTISR[39]	I/O Port P5	P5IFG.x (x = 0 through 7)
INTISR[40]	I/O Port P6	P6IFG.x (x = 0 through 7)
INTISR[41]	Reserved	
INTISR[42]	Reserved	

Aquesta és una part de la taula 6-39 esmentada. A la primera columna tenim un "registre" de 64 bits (no surt complert) que correspon als nostres ISER[0], ICPR[0] els primers 32 bits i ISER[1], ICPR[1] els últims 32 bits.

El que és important és veure que cada bit d'aquest registre està associat amb un vector d'interrupció. Si el posem a "1" al ISER[x] corresponent, l'habilitem al NVIC. Per inhabilitar-la hem de posar a "1" el corresponent bit als registres ICER[x].

Exemple d'Interrupció: vector amb diverses fonts

Si el port 2 el tinguéssim configurat com GPIOs d'entrada que poden generar interrupcions:

```
void PORT2_IRQHandler (void)  //interrupcions GPIO del port 2.
{
    uint8_t flag = P2IV
    P2IE &= 0x00;    //Inhabilitem temporalment les interrupcions de tot el port 2

    switch (flag) {
        case 0x02:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.0
                    break;
        case 0x04:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.1
                    break;
        case 0x06:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.2
                    break;
        case 0x08:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.3
                    break;
        case 0x0A:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.4
                    break;
        case 0x0C:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.5
                    break;
        case 0x0E:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.6
                    break;
        case 0x10:
    // Aquí posem el que volem fer si s'ha generat una interrupció al P2.7
                    break;
        default:    break;
                    }
    P2IE |= 0xFF;    //Tornem a habilitar les interrupcions del port 2
}
```

Figure 10-1. PxIV Register

15	14	13	12	11	10	9	8
Reserved							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
Reserved			PxIV				
r0	r0	r0	r-0	r-0	r-0	r-0	r0

Table 10-4. PxIV Register Description

Bit	Field	Type	Reset	Description
15-5	Reserved	R	0h	Reserved. Reads return 0h
4-0	PxIV	R	0h	Port x interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Port x.0 interrupt; Interrupt Flag: PxIFG.0; Interrupt Priority: Highest 04h = Interrupt Source: Port x.1 interrupt; Interrupt Flag: PxIFG.1 06h = Interrupt Source: Port x.2 interrupt; Interrupt Flag: PxIFG.2 08h = Interrupt Source: Port x.3 interrupt; Interrupt Flag: PxIFG.3 0Ah = Interrupt Source: Port x.4 interrupt; Interrupt Flag: PxIFG.4 0Ch = Interrupt Source: Port x.5 interrupt; Interrupt Flag: PxIFG.5 0Eh = Interrupt Source: Port x.6 interrupt; Interrupt Flag: PxIFG.6 10h = Interrupt Source: Port x.7 interrupt; Interrupt Flag: PxIFG.7; Interrupt Priority: Lowest

Sistema de *Timers* del MSP432P401

Disposem dels següents recursos:

- *Timer* TA0
- *Timer* TA1
- *Timer* TA2
- *Timer* TA3

Fins 16 bits, poden treballar com *PWM*, Repetitiu, Sortir a pins, *Capture/Compare*, Generar Interrupcions…

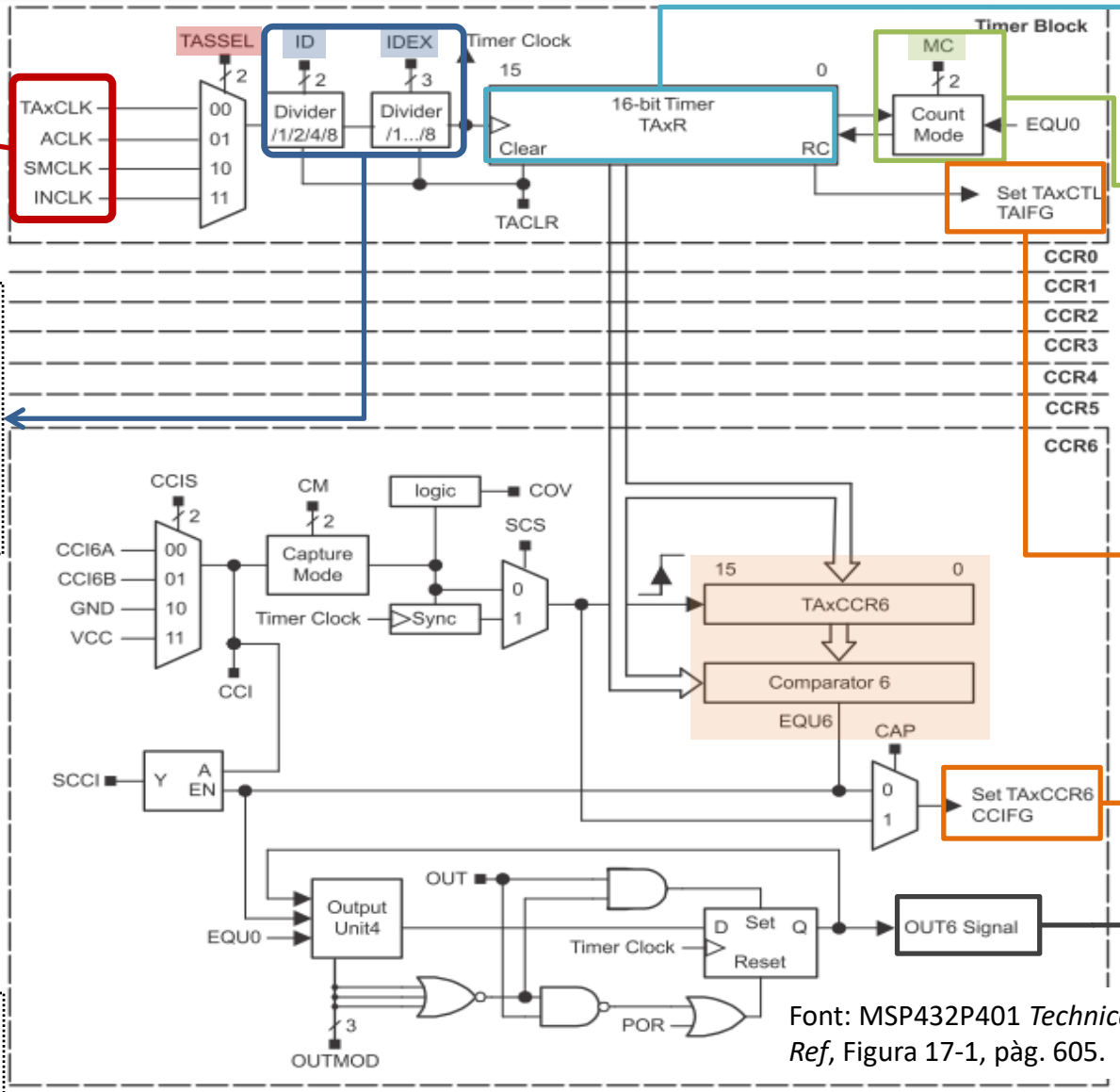
- 2 Timers de 32 bits. (poden generar interrupcions).
- RTC_C (*Real Time Clock*) que pot treballar com:
  - Rellotge en temps real amb funcions de calendari (compensació de mesos de diferents número de dies) i alarmes.
  - Pot generar interrupcions.

Font: MSP432P401 *Datasheet*, capítol 6.9 *Peripherals*

Diagrama de Blocs del *Timers* de 16 bits

Fonts de rellotge, es seleccionen mitjançant **TAxCTL**

Abans d'entrar al *Timer* es pot dividir la font:
ID a **TAxCTL**
IDEX a **TAxEX0**



Font: MSP432P401 *Technical Ref*, Figura 17-1, pàg. 605.

Valors per omissió:
ACLK : 32.768Hz
SMCLK : 2MHz

Aquest és el comptador que es va incrementant amb la font seleccionada **TAxR**

Selecció del Mode de funcionament **TAxCTL**

Podem fer que es generi una interrupció quan el comptador torna a 0 després d'arribar al màxim comptatge **TAxR** (*flag* **TAIFG** de **TAxCTL**)
O carregar un valor diferent al registre de captura comparació que volem fer servir **TAxCCRx** i generar interrupcions al arribar a aquest valor (*flag* **CCIFG** de **TAxCCTLx**)

També podem triar que surti un senyal per un pin del microcontrolador.

Modes de Funcionament dels *Timers* de 16 bits

Els ***Timers* Ax** tenem 4 modes de funcionament que es controlen mitjançant els bits MC del registre TA0CTL, TA1CTL, TA2CTL, TA3CTL, segons el *timer*:

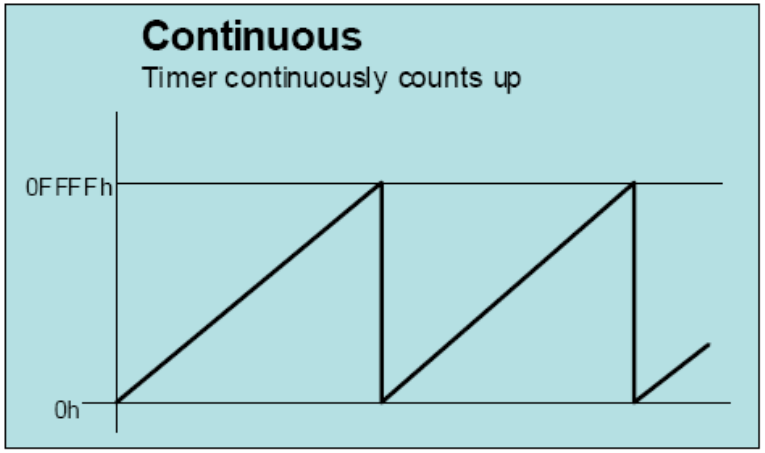
- ***Stop:*** MC=00 (**MC_0**) El *timer* està parat.
- ***Up:*** MC=01 (**MC_1**) El *timer* compta de forma cíclica des de 0 a **TAxCCR0***.
- ***Continous:*** MC=10 (**MC_2**) El *timer* compta de forma cíclica des de 0 al seu valor màxim.
- ***Up/Down:*** MC=11 (**MC_3**) El *timer* compta de forma cíclica des de 0 a **TAxCCR0*** i torna a 0.

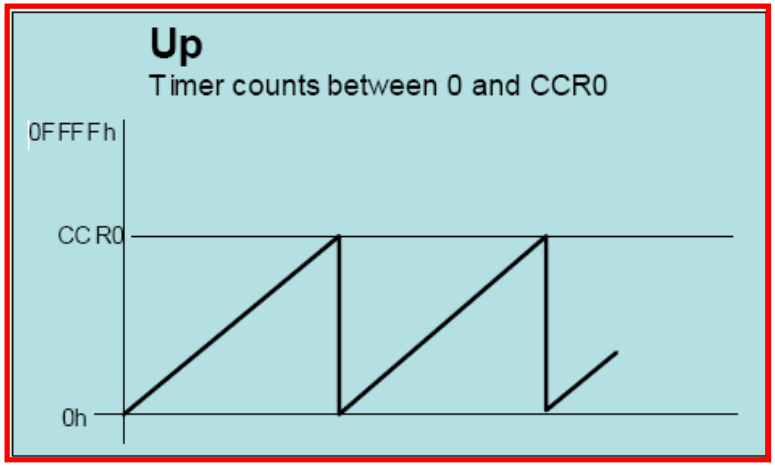
A qualsevol dels modes podem fer que generi una interrupció cada cicle.

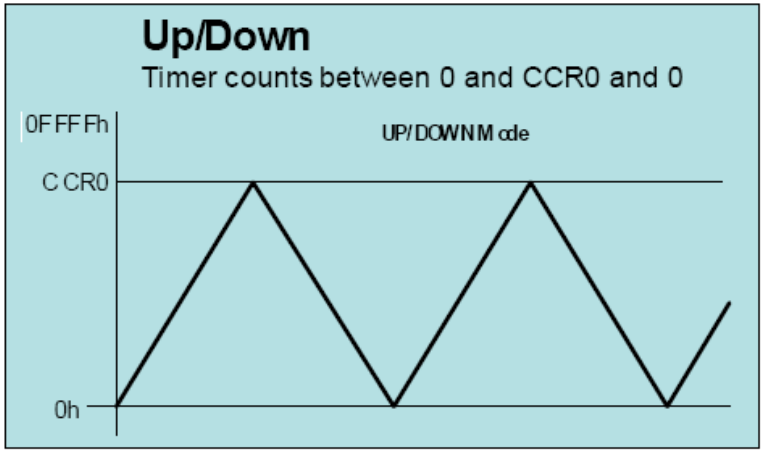
*** TA0CCR0, TA1CCR0, TA2CCR0, TA3CCR0 segons treballem amb TA0 , TA1, TA2 o TA3 respectivament.**

Font: MSP432P4xx *Technical reference*, 17.2.3 pàg. 607-610.

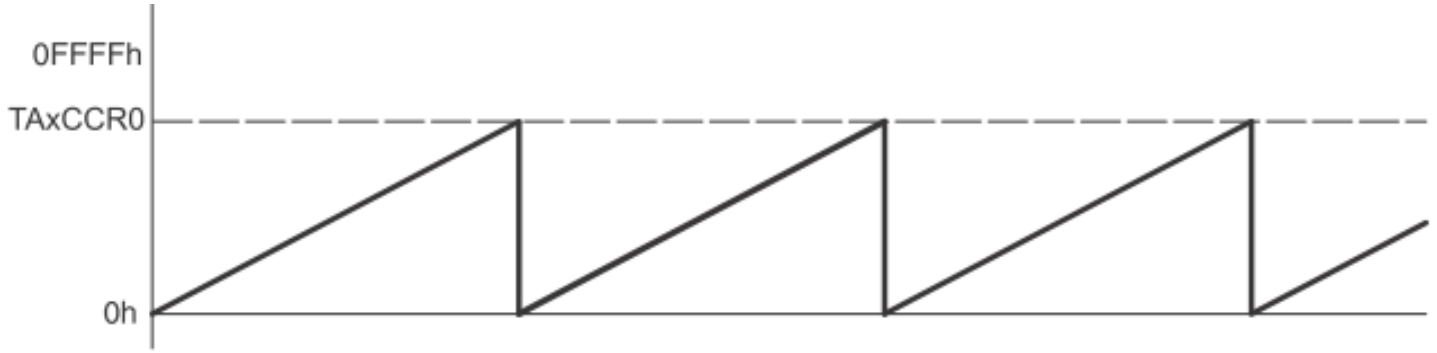
Modes de Funcionament dels *Timers* de 16 bits



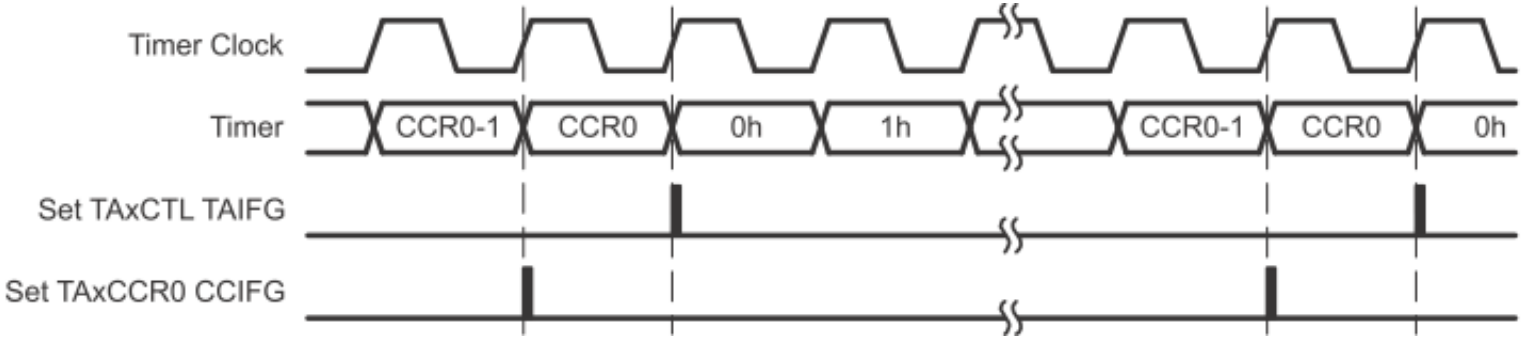




Exemple Mode *Up* al *timers* TA



Evolució d'un *Timer* A en mode Up



Si hem habilitat les interrupcions es poden generar interrupcions amb 2 *flags* diferents el del tornada a 0 del *timer* (TAxIFG) i/o el de arribar a TAxCCR0. De fet, es poden generar més si fem servir els altres registres *capture/compare* del *timer* (veure *Technical reference*).

Font: MSP432P4xx *Technical reference*, 17.2.3 pàg. 607-610.

Interrupcions als *Timers* de 16 bits

Cada *timer* (**TA0, TA1, TA2 i TA3**) té associat 2 vectors d'interrupció, per tant al nostre microcontrolador tenim 8 vectors d'interrupció:

- **Vector d'Interrupció associat a *TAxCCR0***, *flag* **CCIFG** de ***TAxCCTL0***. Per fer servir aquesta font d'interrupció s'ha d'activar al registre ***TAxCCTL0*** (bit **CCIE**), i al NVIC.
- **Vector d'Interrupció *TAxIV*** per la resta de *flags* **CCIFGn** i el **TAxIFG** (*overflow*). Aquestes les activem al registre ***TAxCTL*** (bit **TAIE**), i al NIVC.

Per tant, quan es genera una interrupció d'aquestes, les rutines d'atenció a les interrupcions presenten dues situacions:

- **TA0_0_IRQHandler, TA1_0_IRQHandler, TA2_0_IRQHandler, TA3_0_IRQHandler:** Directes.
- **TA0_N_IRQHandler, TA1_N_IRQHandler, TA2_N_IRQHandler, TA3_N_IRQHandler:** S'han de comprovar els diferents *flags* per saber qui ha generat la interrupció.

Interrupcions als *Timers* de 16 bits

*** Cas del *timer* A0, si és el A1, A2 o A3 això canvia pel nom corresponent**

Com les fem servir:

- Cas *Timer A0, Timer A1, Timer A2 i Timer A3* (associats a CCR0): **Directes**.

```
void TA0_0_IRQHandler (void)*        /Cas del TA0. Aquest és el nom important
{
   TA0CCTL0 &= ~CCIE; //Convé inhabilitar la interrupció al començament
      /* El que volem fer a la rutina d'atenció d'Interrupció */
      /* Aquí no hem de fer cap comprovació addicional ja que */
      /* només pot haver una causa per generar la interrupció */
      /* que el timer corresponent ha arribat al valor de CCR0 programat */
   TA0CCTL0 &= ~CCIFG; //Hem de netejar el flag de la interrupció
   TA0CCTL0 |= CCIE; //S'ha d'habilitar la interrupció abans de sortir
}
```

- Cas *Timer A0, Timer A1, Timer A2 i Timer A3 (CCR1-CCR6 i TAxIFG)*: **Indirectes.**

```
void TA0_N_IRQHandler (void)*        /Cas del TA0. Aquest és el nom important
{
      /* Analitzar els flags CCIFGx i TxIFG per saber qui ha demanat la interrupció,
         llavors fer que el corresponent a cadascuna.
         Com al cas dels Ports, si fem servir TAxIV, l'hem de guardar a una variable
         ja que quan accedim al registre es neteja automàticament */
}
```

Cas Timer A0 Vector Múltiple

```
void TA0_N_IRQHandler(void);
{
  uint16_t flag = TA0IV;
  TA0CTL &= ~(TAIE);
  switch (flag)
  {
    case  0:          // No interrupt
                break;
    case  2:          // TA0CCR1
                break;
    case  4:          // TA0CCR2
                break;
    case  6:          // TA0CCR3
                break;
    case  8:          // TA0CCR4
                break;
    case 10:          // TA0CCR5
                break;
    case 12:          // TA0CCR6
                break;
    case 14:          // TA0IFG overflow handler
                 break;
  }
  TA0CTL |= TAIE;
}
```

VECTOR D'INTERRUPCIÓ: TAxIV

Figure 17-19. TAxIV Register

15	14	13	12	11	10	9	8
TAIV							
r0	r0	r0	r0	r0	r0	r0	r0
7	6	5	4	3	2	1	0
TAIV							
r0	r0	r0	r0	r-0	r-0	r-0	r0

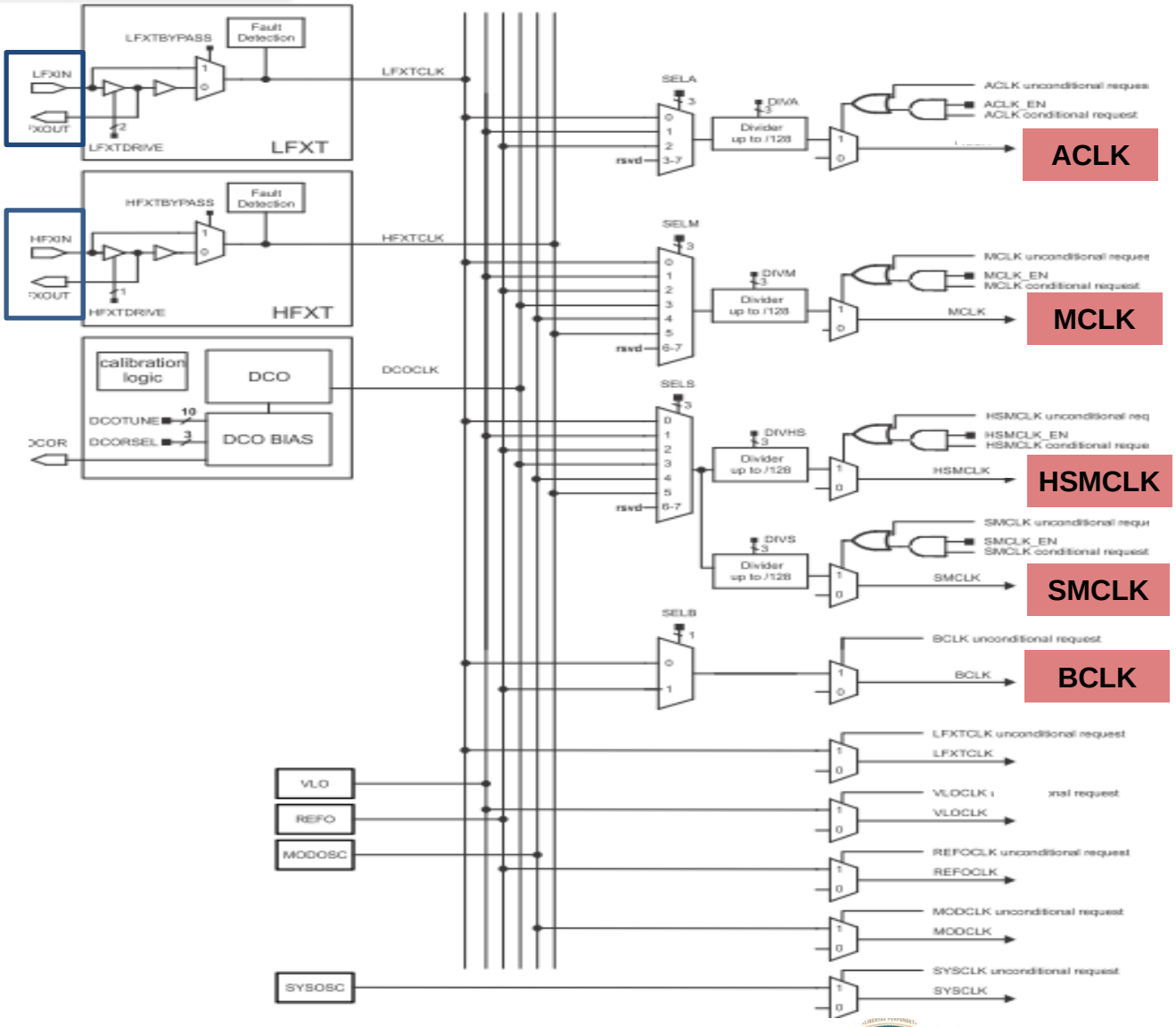
Table 17-8. TAxIV Register Description

Bit	Field	Type	Reset	Description
15-0	TAIV	R	0h	Timer_A interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Capture/compare 1; Interrupt Flag: TAxCCR1 CCIFG; Interrupt Priority: Highest 04h = Interrupt Source: Capture/compare 2; Interrupt Flag: TAxCCR2 CCIFG 06h = Interrupt Source: Capture/compare 3; Interrupt Flag: TAxCCR3 CCIFG 08h = Interrupt Source: Capture/compare 4; Interrupt Flag: TAxCCR4 CCIFG 0Ah = Interrupt Source: Capture/compare 5; Interrupt Flag: TAxCCR5 CCIFG 0Ch = Interrupt Source: Capture/compare 6; Interrupt Flag: TAxCCR6 CCIFG 0Eh = Interrupt Source: Timer overflow; Interrupt Flag: TAxCTL TAIFG; Interrupt Priority: Lowest

MSP432P401 *Technical reference*, 17.3 pàg. 622

Sistema de Rellotge CS*

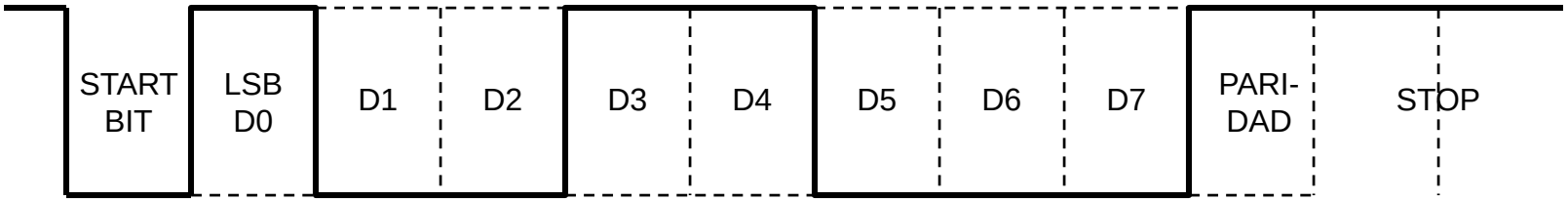
- **Diverses fonts de rellotge "independents":**
  - ***Low Frequency***
    - **LFXTCLK Low Freq (Extern rang de 32kHz o menys)**
    - **VLOCLK 10 kHz (Intern, molt baixa potència)**
    - **REFOCLK 32768 Hz (Intern LF)**
  - ***High Frequency***
    - **HFXTCLK High-Freq (Extern 1MHz-48MHz)**
    - **DCOCLK Oscil·lador intern programable (3MHz per defecte)**
    - **MODCLK intern 25MHz**
    - **SYSOSC intern 5MHz**
- **Fonts de rellotge disponibles pels dispositius:**
  - **ACLK Auxiliary Clock. Es pot seleccionar a partir de les fonts *LowFreq* anteriors.**
  - **MCLK Master clock (CPU)**
  - **HSMCLK Subsystem master clock**
  - **SMCLK Secondary master clock**
  - **BCLK Low-speed backup clock.**
- **Els Rellotges funcionen quan es necessiten.**



**** CS = Clock System***

Comunicacions Sèrie UART (RS232)

- BUS SERIE (mínim 2 fils i GND).
- ASÍNCRON (el transmissor i el receptor tenen rellotges diferents).
- El sincronisme es realitza per cada caràcter transmès.
- FORMAT: de 7 a 12 bits per caràcter a transmetre.
  - 1 bit de START.
  - De 5 a 8 bits d'INFORMACIÓ.
  - 1 bit de PARITAT (opcional).
  - 1 o 2 bits de STOP.



Comunicacions Sèrie UART (RS232)

- UART sampling vs oversampling

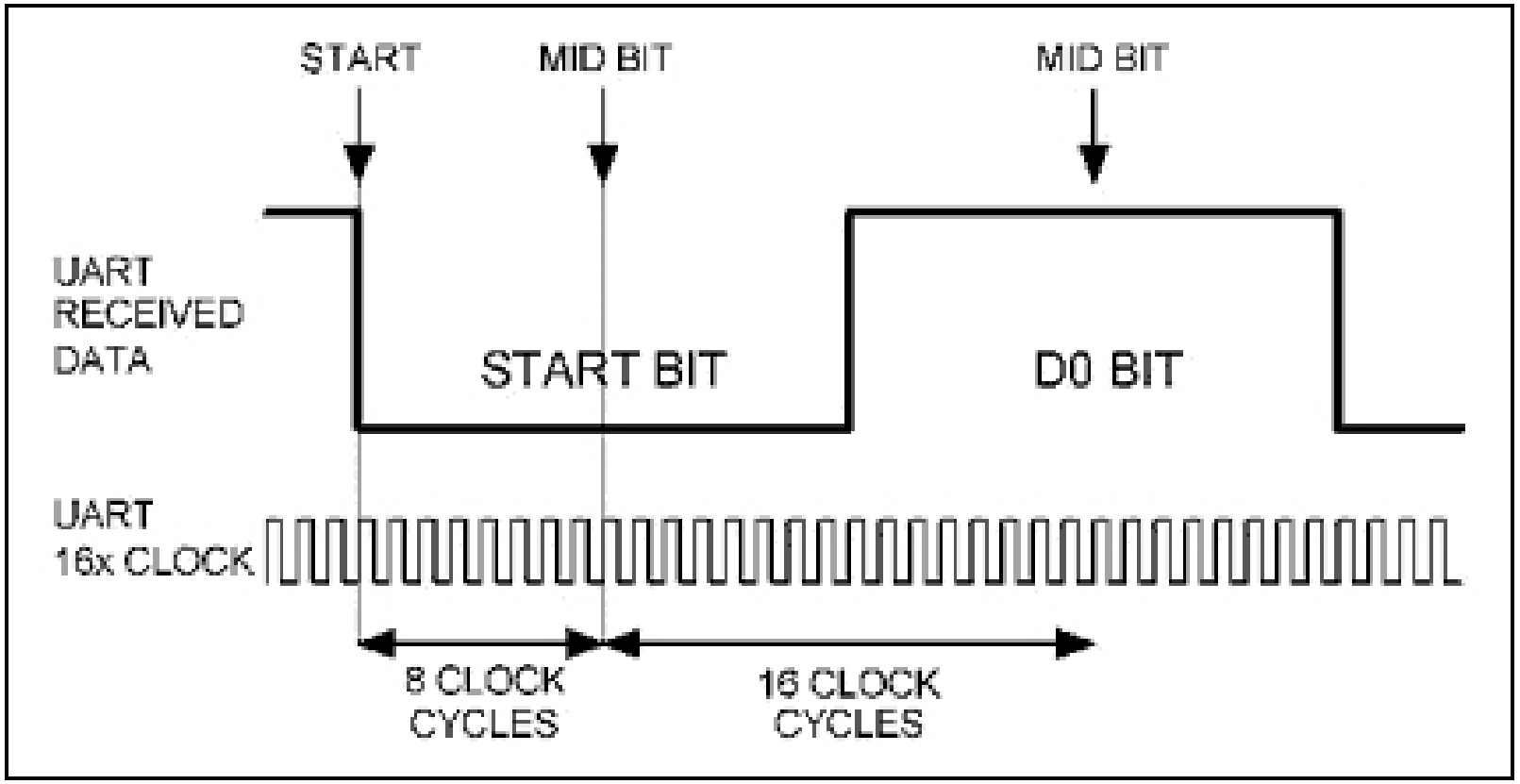
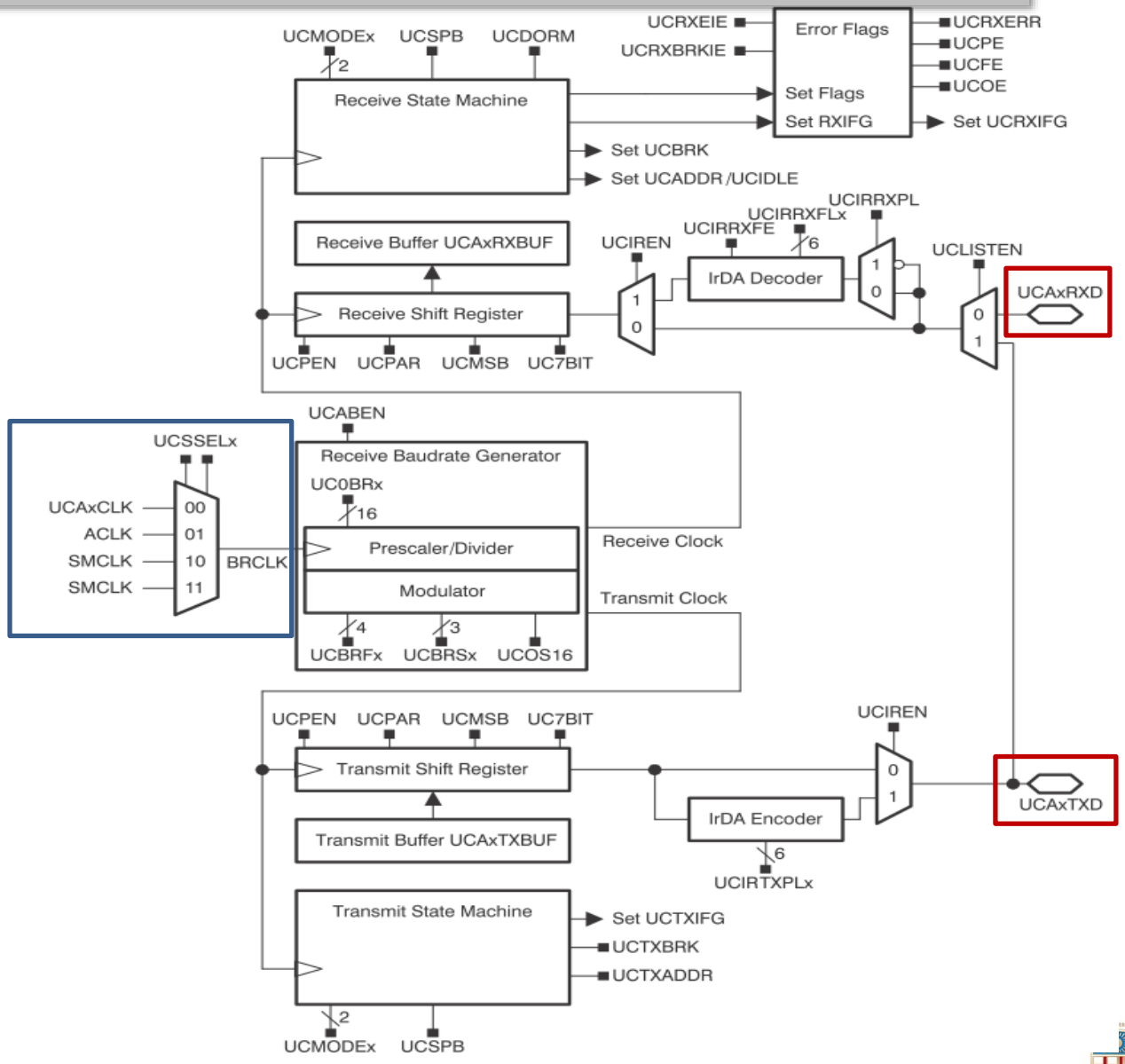
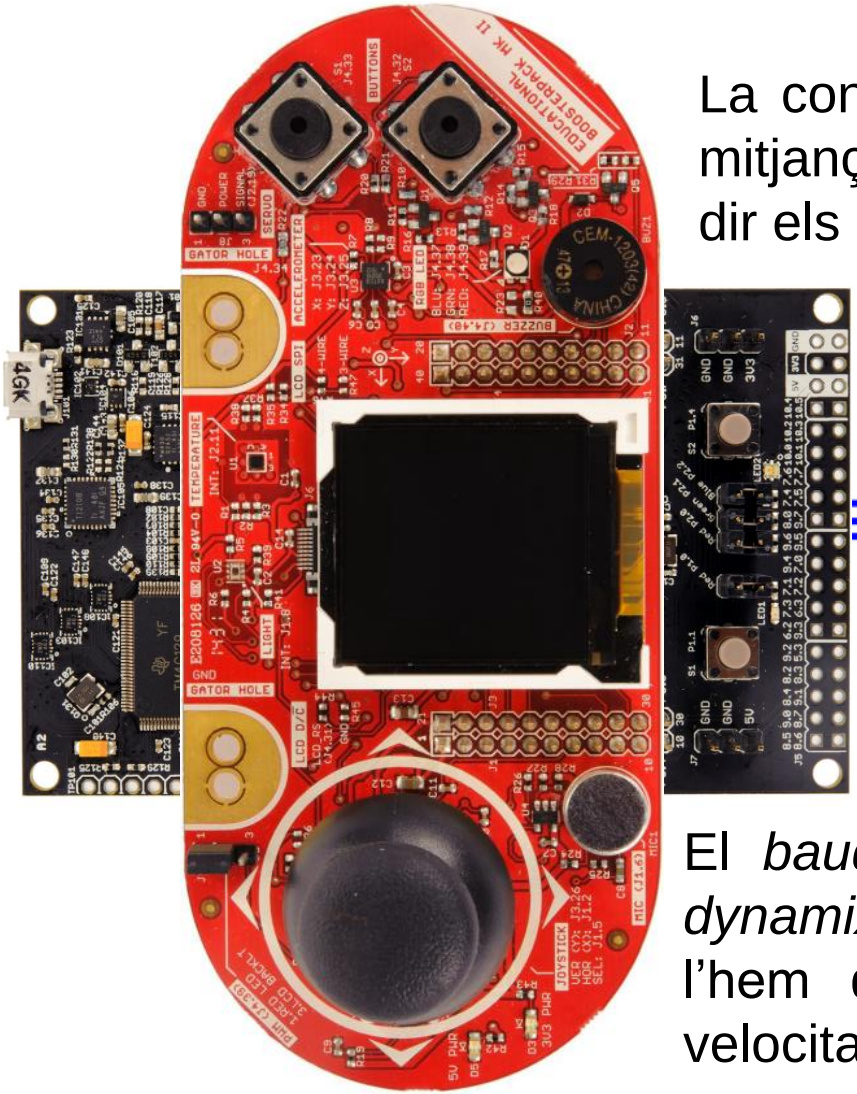


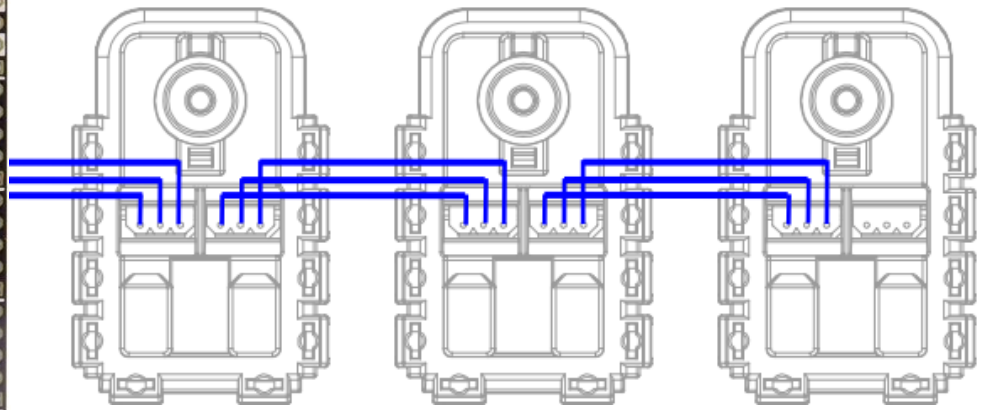
Diagrama de blocs de la eUART del MSP432P401



Connexió UART amb els Mòduls Dynamixel



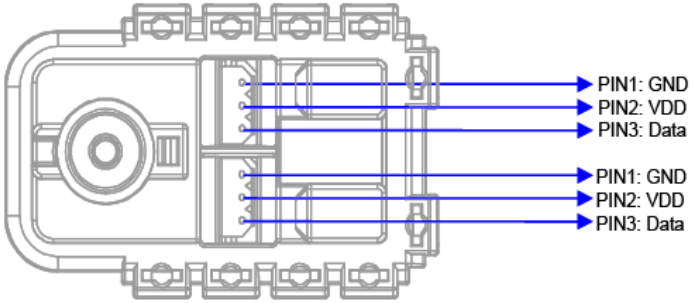
La connexió entre la placa i els mòduls la fem mitjançant una UART A2 del port 3. I com vàrem dir els mòduls es connecten en *daisy chain*.



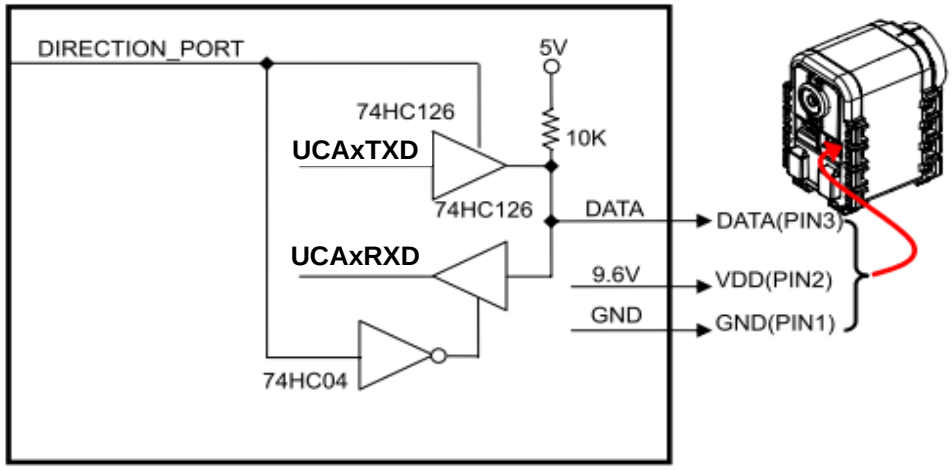
El *baud rate* que hem programat als mòduls *dynamixel* és de 500kb/s, i per tant la UART l'hem de programar per treballar a aquesta velocitat.

Connexió UART amb els Mòduls Dynamixel

El segon punt a tenir en compte és que els mòduls *dynamixel* fan servir comunicació asíncrona però *Half-duplex* (només té una línia "Data" per transmetre i rebre) mentre que la UART en principi és *Full-duplex* (té una línia per transmetre UCAxTXD i una per rebre UCAxRXD).



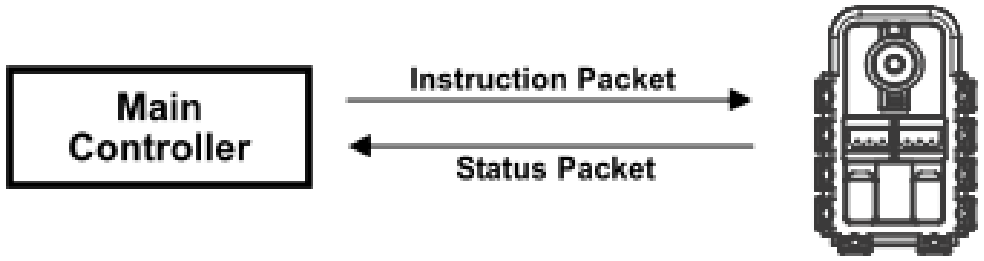
Per solucionar-lo fem un circuit que passa de dues línies a una i viceversa:



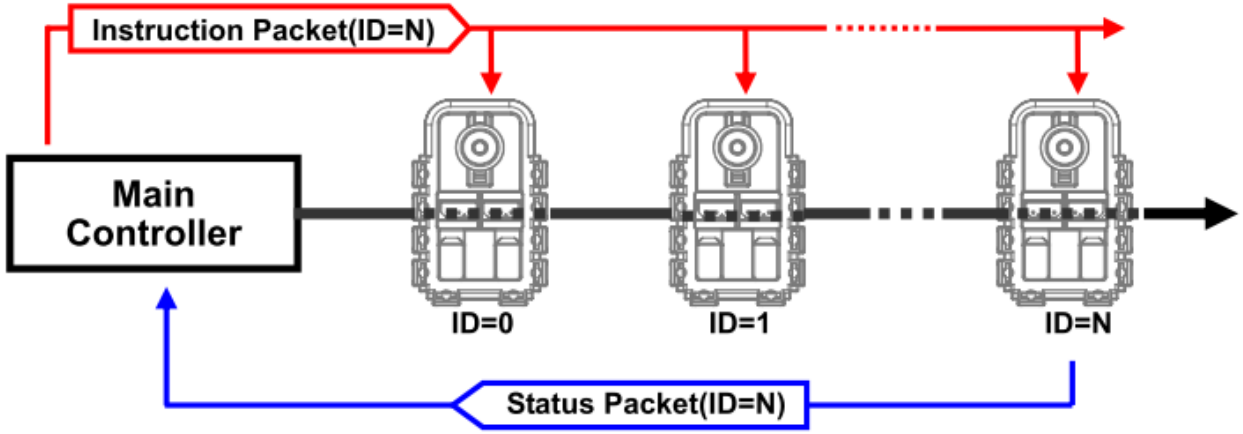
Hem de tenir en compte que des del microcontrolador, per programa, hem de controlar el senyal "*DIRECTION_PORT*". A la nostra placa l'hem connectat al *port* 3, al *pin* 3.0. Hem d'inicialitzar-lo com GPIO de sortida, i gestionar el senyal en funció de si volem enviar o rebre missatges.

Protocol Comunicació dels Mòduls del Robot

Paquets: El microcontrolador envia un "*Instruction Packet*" (format per diversos bytes) al mòdul robot i aquest li contesta amb un "*Status Packet*".



Al "*Instruction Packet*" hi ha un paràmetre que és l'Identificador (ID) del mòdul al que va dirigit, només aquest mòdul rebrà les dades que s'envien. L'ID ha de ser únic, no pot repetir-se a cap altre mòdul.



Mapa de memòria DYNAMIXEL

No-volàtil (EEPROM)

Address	Item	Access	Initial Value
0(0X00)	Model Number(L)	RD	12(0x0C)
1(0X01)	Model Number(H)	RD	0(0x00)
2(0X02)	Version of Firmware	RD	?
3(0X03)	ID	RD,WR	1(0x01)
4(0X04)	Baud Rate	RD,WR	1(0x01)
5(0X05)	Return Delay Time	RD,WR	250(0xFA)
6(0X06)	CW Angle Limit(L)	RD,WR	0(0x00)
7(0X07)	CW Angle Limit(H)	RD,WR	0(0x00)
8(0X08)	CCW Angle Limit(L)	RD,WR	255(0xFF)
9(0X09)	CCW Angle Limit(H)	RD,WR	3(0x03)
10(0x0A)	(Reserved)	-	0(0x00)
11(0X0B)	the Highest Limit Temperature	RD,WR	85(0x55)
12(0X0C)	the Lowest Limit Voltage	RD,WR	60(0X3C)
13(0X0D)	the Highest Limit Voltage	RD,WR	190(0xBE)
14(0X0E)	Max Torque(L)	RD,WR	255(0XFF)
15(0X0F)	Max Torque(H)	RD,WR	3(0x03)
16(0X10)	Status Return Level	RD,WR	2(0x02)
17(0X11)	Alarm LED	RD,WR	4(0x04)
18(0X12)	Alarm Shutdown	RD,WR	4(0x04)
19(0X13)	(Reserved)	RD,WR	0(0x00)
20(0X14)	Down Calibration(L)	RD	?
21(0X15)	Down Calibration(H)	RD	?
22(0X16)	Up Calibration(L)	RD	?
23(0X17)	Up Calibration(H)	RD	?

Volàtil (RAM)

24(0X18)	Torque Enable	RD,WR	0(0x00)
25(0X19)	LED	RD,WR	0(0x00)
26(0X1A)	CW Compliance Margin	RD,WR	0(0x00)
27(0X1B)	CCW Compliance Margin	RD,WR	0(0x00)
28(0X1C)	CW Compliance Slope	RD,WR	32(0x20)
29(0X1D)	CCW Compliance Slope	RD,WR	32(0x20)
30(0X1E)	Goal Position(L)	RD,WR	[Addr36]value
31(0X1F)	Goal Position(H)	RD,WR	[Addr37]value
32(0X20)	Moving Speed(L)	RD,WR	0
33(0X21)	Moving Speed(H)	RD,WR	0
34(0X22)	Torque Limit(L)	RD,WR	[Addr14] value
35(0X23)	Torque Limit(H)	RD,WR	[Addr15] value
36(0X24)	Present Position(L)	RD	?
37(0X25)	Present Position(H)	RD	?
38(0X26)	Present Speed(L)	RD	?
39(0X27)	Present Speed(H)	RD	?
40(0X28)	Present Load(L)	RD	?
41(0X29)	Present Load(H)	RD	?
42(0X2A)	Present Voltage	RD	?
43(0X2B)	Present Temperature	RD	?
44(0X2C)	Registered Instruction	RD,WR	0(0x00)
45(0X2D)	(Reserved)	-	0(0x00)
46[0x2E)	Moving	RD	0(0x00)
47[0x2F)	Lock	RD,WR	0(0x00)
48[0x30)	Punch(L)	RD,WR	32(0x20)
49[0x31)	Punch(H)	RD,WR	0(0x00)

Instruccions

Instruction	Function	Value	Number of Parameter
PING	No action. Used for obtaining a Status Packet	0x01	0
READ DATA	Reading values in the Control Table	0x02	2
WRITE DATA	Writing values to the Control Table	0x03	2 ~
REG WRITE	Similar to WRITE_DATA, but stays in standby mode until the ACION instruction is given	0x04	2 ~
ACTION	Triggers the action registered by the REG_WRITE instruction	0x05	0
RESET	Changes the control table values of the Dynamixel actuator to the Factory Default Value settings	0x06	0
SYNC WRITE	Used for controlling many Dynamixel actuators at the same time	0x83	4~

Protocol Comunicació dels Mòduls del Robot

INSTRUCTION PACKET

Format dels Paquets: seqüència de bytes enviat pel microcontrolador

OXFF	0XFF	ID	LENGTH	INSTRUCTION	PARAMETER1	…	PARAMETER N	CHECK SUM

0xFF, 0xFF: Indiquen el començament d'una trama.

ID: Identificador únic de cada mòdul *Dynamixel* (entre 0x00 i 0xFD).
El identificador 0xFE és un "*Broadcasting ID*" que van a tots els mòduls (aquests no retornaran *Status Packet*)

LENGTH: El número de bytes del paquet (trama) = Nombre de paràmetres + 2

INSTRUCTION: La instrucció que se li envia al mòdul.

PARAMETER 1...N: No sempre hi ha paràmetres, però hi ha instruccions que si necessiten.

CHECK SUM: Paràmetre per detectar possibles errors del comunicació, es calcula així:

Check Sum = ~(ID + LENGTH + INSTRUCTION + PARAMETER 1 + ... + PARAMETER N)

Protocol Comunicació dels Mòduls del Robot

STATUS PACKET

Format dels Paquets: seqüència de bytes amb que respon el mòdul

OXFF	0XFF	ID	LENGTH	ERROR	PARAMETER1	PARAMETER2	…	PARAMETER N	CHECK SUM

0xFF, 0xFF: Indiquen el començament d'una trama.

ID: Identificador del mòdul.

LENGTH: El número de bytes del paquet.

ERROR:

Bit	Name	Details
Bit 7	0	-
Bit 6	Instruction Error	Set to 1 if an undefined instruction is sent or an action instruction is sent without a Reg_Write instruction.
Bit 5	Overload Error	Set to 1 if the specified maximum torque can't control the applied load.
Bit 4	Checksum Error	Set to 1 if the checksum of the instruction packet is incorrect.
Bit 3	Range Error	Set to 1 if the instruction sent is out of the defined range.
Bit 2	Overheating Error	Set to 1 if the internal temperature of the Dynamixel unit is above the operating temperature range as defined in the control table.
Bit 1	Angle Limit Error	Set as 1 if the Goal Position is set outside of the range between CW Angle Limit and CCW Angle Limit.
Bit 0	Input Voltage Error	Set to 1 if the voltage is out of the operating voltage range as defined in the control table.

PARAMETER 1...N: Si es necessiten.

CHECK SUM: Paràmetre per detectar possibles errors del comunicació, es calcula així:

Check Sum = ~(ID + LENGTH + ERROR + PARAMETER 1 + ... + PARAMETER N)

Exemple

Example 16	**Turn on the LED and Enable Torque for a Dynamixel actuator with an ID of 0**
Instruction Packet	Instruction = WRITE_DATA, Address = 0x18, DATA = 0x01, 0x01
Communication	->[Dynamixel]:FF FF 00 05 03 18 01 01 DD (LEN:009) <-[Dynamixel]:FF FF 00 02 00 FD (LEN:006)
Status Packet Result	NO ERROR

Mutex

A Mutex allows guaranteeing that only ONE thread at a given time is allowed to executed a certain code area, i.e., it is a **locking mechanism**.

```
#include <pthread.h>

        pthread_mutex_lock(pthread_mutex_t* mutex);
        //Only one thread may execute this code area at the same time
        cnt++;
        pthread_mutex_unlock(pthread_mutex_t* mutex);
```

Hence, adding a mutex lock around the *cnt*++ operation of the previous example will ensure that only one thread will be able to perform this operation while the other thread will be **blocked** waiting until the first one finished.

Semaphores

A semaphore is like a **counter variable** on which the increment or decrement operations are guaranteed to be **atomic**. It's most used for **signaling**.

A binary semaphore will have only to possible values, 0 and 1.

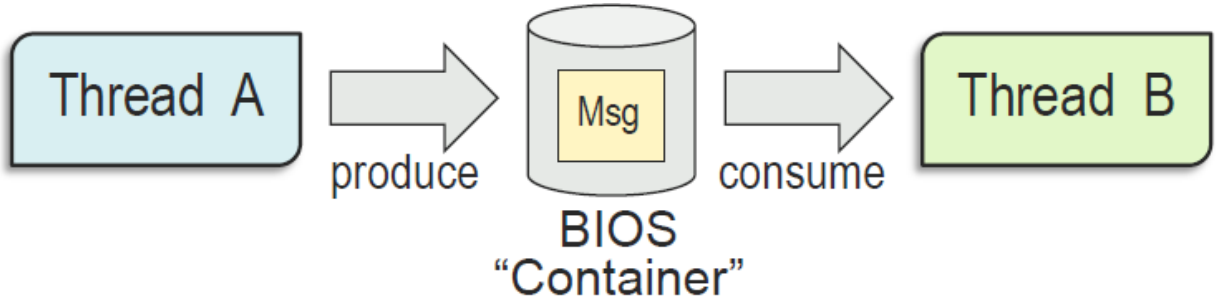
We may also change the previous code so that both threads before executing the statements with the shared variable will check if the semaphore is 1 (green light). If it is, they will decrement it to zero (red light) and go on with their code. And if it is already zero, there will wait until it is one again (the thread will be blocked).

This way, only one thread at time will be able to modify the variable *cnt*.

```
#include <semaphore.h>

        sem_wait(sem_t* semaphore);
        consumer(&data);
```

Thread memory sharing



How it Works:

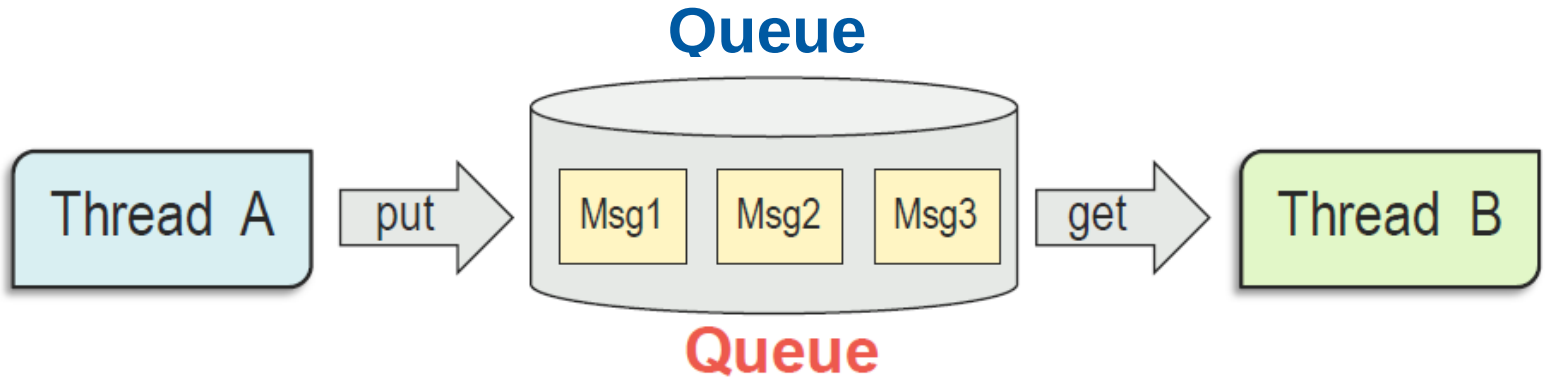
- Data is UNIDIRECTIONAL – one thread waits for the other to ***produce***
- Thread B often BLOCKS until data is ***produced***

Advantages:

- Both threads use same protocol – contention is often avoided
- Threads become more "modular" and therefore reuse improves

Examples:

- BIOS: **Queue, Mailbox**
- May have built-in synchronization or user can add signaling via *Semaphores* and *Events*



- A **Queue** is a BIOS object that can contain *anything* you like
- "Data" is called a "Msg" – simply a structure defined by the user
- Msgs are "reclaimed" on a FIFO basis
- Key APIs:

```
Queue_put();
Queue_get();
```

- *Advantages*: simple, not copy based
- *Disadvantage*: no signaling built in