Programació de sistemes encastats

Un altre llibre sobre el mateix

Màrius Montón, PhD

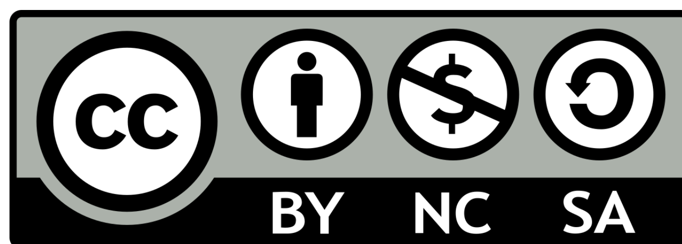
Aquesta pàgina està en blanc expressament, tot va bé.

Copyright © 2018, 2019 Màrius Montón

Versió: 1.0

Data: 8 de novembre de 2020

Distribuit segons la llicència Creative Commons Reconeixement-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) (link al text de la llicència)



Sou lliure de:

Compartir — copiar i redistribuir el material en qualsevol mitjà i format
Adaptar — remesclar, transformar i crear a partir del material

Amb els termes següents:

Reconeixement — Heu de reconèixer l'autoria de manera apropiada, proporcionar un enllaç a la llicència i indicar si heu fet algun canvi. Podeu fer-ho de qualsevol manera raonable, però no d'una manera que suggereixi que el llicenciador us dóna suport o patrocina l'ús que en feu.
NoComercial — No podeu utilitzar el material per a finalitats comercials.
CompartirIgual — Si remescleu, transformeu o creeu a partir del material, heu de difondre les vostres creacions amb la mateixa llicència que l'obra original.

Latex Template based on The Legrand Orange Book by Mathias Legrand.

Aquesta pàgina està en blanc expressament, tot va bé.

A tots els meus alumnes, que m'han ajudat a entendre i aprendre.

Màrius

Vull agrair a Aitor Mejias, Borja Martínez, Lluís Gesa, Cristina Cano, Francisco Vázquez, Pere Tuset, Xose Pérez, Jordi Binefa, Xavier Fitó i Lluís Ribas pels comentaris i correccions fetes al text. Com de costum tots els errors que encara hi hagi al text i al codi son cosa meva i no pas seva.

Màrius

Aquesta pàgina està en blanc expressament, tot va bé.

Índex

I Nocions bàsiques

1 Introducció 13

2 Breu introducció als sistemes encastats 19

II Programació de perifèrics I

3 Consola de Debug 31

4 Fent servir printf 33

5 Gestió de rellotges 37

6 GPIO 41

7 Controlador d’interrupcions 47

8 Timers 51

9 RTC 57

10 PWM 61

11 *Watchdog* 69

III Programació de perifèrics II

12 ADC . . . 75
13 DAC . . . 79
14 UART . . . 85
15 I2C . . . 89
16 SPI . . . 93
17 DMA . . . 95
18 FLASH . . . 101
19 Mòduls criptogràfics . . . 105
20 Altres perifèrics . . . 109
21 Una aplicació completa . . . 117

IV FreeRTOS

22 Conceptes bàsics de FreeRTOS . . . 127
23 Primer exemple amb FreeRTOS . . . 137
24 Controlant el temps a les tasques . . . 139
25 Comunicació entre tasques . . . 143
26 Exemple amb la UART i interrupcions . . . 159
27 Una aplicació completa amb FreeRTOS . . . 163
28 Ús del watchdog en RTOS . . . 167
29 *Drivers* en multi-tasca . . . 169

V Models de programació

30 Model d'interfície amb perifèrics . . . 177
31 Models de computació . . . 179

32 Tractament del temps 191

VI Temes avançats

33 Gestió d'excepcions 199

34 *Shadow Registers* 203

35 Baix cosum 205

36 Documentant el codi 211

37 CMSIS 215

38 Normes de codificació 217

39 DSP 221

40 C++ vs C 223

41 Relació Esquemàtic i FW 231

42 Inicialització del sistema i del llenguatge C 235

43 Treballant amb punt flotant 239

VII Bones pràctiques

44 Ús de memòria dinàmica 247

45 Ús de *volatile* 249

46 Funcions re-entrants 251

47 *Deadlock* 253

48 Inversió de prioritats 255

49 Assignació de prioritats 257

50 Mida de les cues 259

51 *Debounce* 263

52 Ús eficient de printf 267

53 Empaquetant estructures 269

VIII Índex, Bibliografia, Glossari

Enllaços dels exemples 277

Bibliografia 279

Glossari 287

Acrònims 291

Índex de figures 297

Índex de llistats 299

Índex de funcions 303

Nocions bàsiques

1 Introducció 13

1.1 El que aquest llibre és

1.2 El que aquest llibre no és

1.3 Material per seguir el curs

1.4 Eines

2 Breu introducció als sistemes encastats 19

2.1 Microcontroladors

2.2 ARM Cortex

2.3 Arquitectura

2.4 Rapidesa d’un microcontrolador

Aquesta pàgina està en blanc expressament, tot va bé.

1. Introducció

Els textos d'aquest llibre es basen en les publicacions de l'autor al blog publicat durant el 2017 i 2018 (`https://sistemesencastats.wordpress.com`). El llibre en si està publicat com a codi obert a l'adreça `https://github.com/mariusmm/Llibreencastats` on es pot baixar tot el codi LaTeX i generar el pdf un mateix.

Com es comenta en el propi blog, suposem que el lector té coneixements de programació en llenguatge C, coneixements bàsics d'arquitectura de computadors i nocions bàsiques d'electrònica.

Tot el codi dels exemples està publicat amb llicència oberta (GNU GPLv3) [1] al repositori del curs a GitHub (`https://github.com/mariusmm/cursembedded`).

L'estructura del curs serà una breu introducció sobre l'arquitectura del microcontrolador que usarem durant el curs, seguirem amb una descripció dels perifèrics més habituals i un codi d'exemple per cada un. A continuació es detallarà l'ús d'un Sistema Operatiu en temps real per el desenvolupament d'aplicacions complexes, es continuarà amb un capítol dedicat al test en sistemes encastats i per últim un capítol de conceptes avançats.

1.1 El que aquest llibre és

Aquest llibre és per tot aquell que amb coneixements de programació en llenguatge C, coneixements d'arquitectures de computadors, coneixements mínims de HW i electrònica vulgui endinsar-se en la programació de sistemes encastats basats en microcontroladors.

També va dirigit a aquelles persones que tenen experiència amb sistemes tipus Arduino i volen entendre i poder afegir codi o crear noves biblioteques. Tot i que no parlarem específicament d'Arduino a cap part del llibre, els coneixements genèrics serveixen per aquest sistema.

1.2 El que aquest llibre no és

Aquest llibre està pensat per donar una introducció a certs aspectes del disseny i programació de sistemes encastats basats en microcontroladors, des de l'ús de biblioteques de fabricants fins a

Sistemes Operatius en Temps Real.

El que no tracta aquest llibre és de plataformes existents com ara Arduino [2]. Aquesta plataforma, tot i que molt valuosa i que ha popularitzat immensament la programació i l'ús de sistemes encastats al gran públic, gràcies sobretot a la seva senzillesa d'ús i a la enorme quantitat de codi d'exemple i biblioteques, creiem que no és adequada per donar un visió detallada de tots els temes que es volen tractar en aquest llibre. Vist d'una altra manera, l'objectiu d'aquest curs és, entre d'altres, habilitar al lector usuari d'Arduino perquè pugui desenvolupar per si mateix noves biblioteques de baix nivell d'Arduino i entendre com està implementat.

Tampoc és un tractat específic sobre l'*interface* amb sensors i el coneixement profund sobre el tema. Per aquest cas concret, hi ha magnífics llibres amb una descripció exhaustiva sobre el tema, com per exemple aquest [3].

1.3 Material per seguir el curs

Tot seguit es presenta el material necessari per fer les parts pràctiques del curs, això inclou una placa de prototipat i uns pocs components, tots ells de baix preu.

1.3.1 Placa de prototipat

La part pràctica del curs es basa en la placa de prototipat EFM32TG-STK3300 Starter Kit de Silicon Labs. Aquesta placa porta un microcontrolador EFM32TG840F32 amb 32 KB de memòria FLASH i 4 KB de memòria RAM[1] (veure el Reference Manual [4] i més endavant **Secció 2.2 - ARM Cortex**).

Aquesta PCB porta un parell de botons, un LED[2] i un connector on hi ha tot de pins amb diferents funcions que podrem utilitzar quan ho necessitem.

Per treballar amb aquesta plataforma, cal instal·lar el conjunt d'aplicacions **Simplicity Studio** versió 4 [5]. Hi ha versions per Linux, Mac i Windows.

1.3.2 Dispositius auxiliars

Per tenir dades variades i de diferent natura per les aplicacions d'exemple, farem servir els següents dispositius auxiliars:

- Potenciòmetre: aquest component, que és una resistència variable, ens permetrà probar un ADC[3]
- Sensor de llum, color i moviment APDS-9960 [6]. Es pot adquirir a qualsevol botiga on-line ja muntat a una PCB que incorpora uns connectors senzills per connectar-la a la PCB de desenvolupament.
- Cables de connexió tipus Dupont amb connector femella als dos extrems (Figura 1.2).

[1] Random Access Memory
[2] Light Emission Diode
[3] Analog to Digital Converter

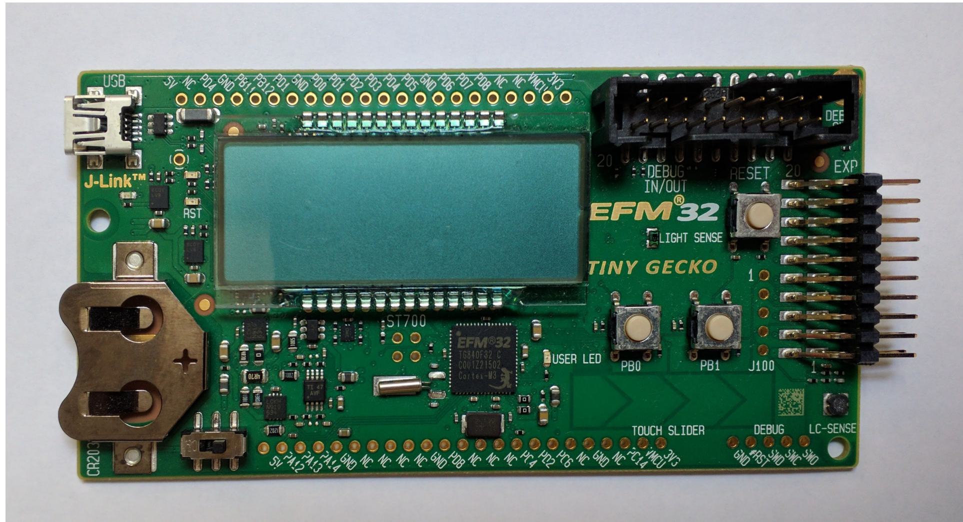


Figura 1.1: Fotografia de la placa de desenvolupament de SiliconLabs

Figura 1.2: Cables dupont

1.4 Eines

Per treballar amb microcontroladors ens calen un conjunt d'eines específiques i algunes de més genèriques. Anem a veure-les amb un cert detall.

1.4.1 Programadors i *debuggers*

Un microcontrolador (també dit MCU[4] o µC) es diferència d'un processador de propòsit general en moltes coses, però una de les diferències més notables és que el propi microcontrolador generalment incorpora la seva pròpia memòria (tanta RAM com ROM[5]) on s'emmagatzema el codi a executar i les variables i dades a tractar. Quan el microcontrolador s'engega comença a executar el codi que troba a la ROM a certa posició.

Per descarregar el fitxer binari a la ROM, cal un dispositiu extern al microcontrolador que emmagatzema el fitxer a la ROM del microcontrolador. Aquests dispositius poden ser totalment externs al nostre circuit, i es coneixen com programadors o darrerament es veuen incorporats a la pròpia PCB i s'hi accedeix via USB. Sigui com sigui, cal aquest programador per gravar la memòria FLASH (que funciona com una ROM pel microcontrolador).

A més, aquest programador sol afegir característiques de *debug*, de manera que podem controlar l'execució del microcontrolador, inspeccionar el valor de variables o posicions de memòria, accedir a la consola de *debug*, etc.

En el cas de la nostra placa de prototipat, aquest programador i *debugger* està integrat a la pròpia PCB, de manera que no ens cal res més que la PCB i un cable USB per programar i *debugger* el microcontrolador sense cap altre dispositiu auxiliar.

1.4.2 *Toolchain*

Com per tot processador, cal un seguit d'eines que ajudin a traduir el nostre codi (normalment C o C++) en instruccions màquina que la CPU pugui processar. Aquestes eines son el compilador i el *linker*. El compilador fa aquesta traducció pròpiament dita i genera fitxers objecte i el *linker* recull tot de fitxers objecte per crear un sol fitxer executable o biblioteca.

En el cas dels microcontroladors, hem d'acabar obtenint un fitxer executable que serà el que el microcontrolador començarà a executar quan s'engegui. Aquest fitxer haurà de tenir tot el conjunt de biblioteques i funcions necessàries per la correcta execució de l'aplicació, ja que en aquest context no tenim cap mena de sistema operatiu que ens proporcioni cap ajuda ni biblioteques.

També és habitual disposar d'algun IDE[6] que ens agrupa totes les eines en un entorn amigable i senzill (veure Figura 1.3).

L'IDE de Simplicity fa servir com a compilador el compilador per ARM de GNU [7]. Aquest compilador és de codi obert i lliure i és àmpliament utilitzant per la majoria de fabricants a les seves eines. Aquest es un compilador creuat, és a dir, es pot executar en un processador diferent del processador pel que està compilant el codi.

[4] *MicroController Unit*
[5] Read-Only Memory
[6] *Integrated development environment*, Entorn integrat de desenvolupament

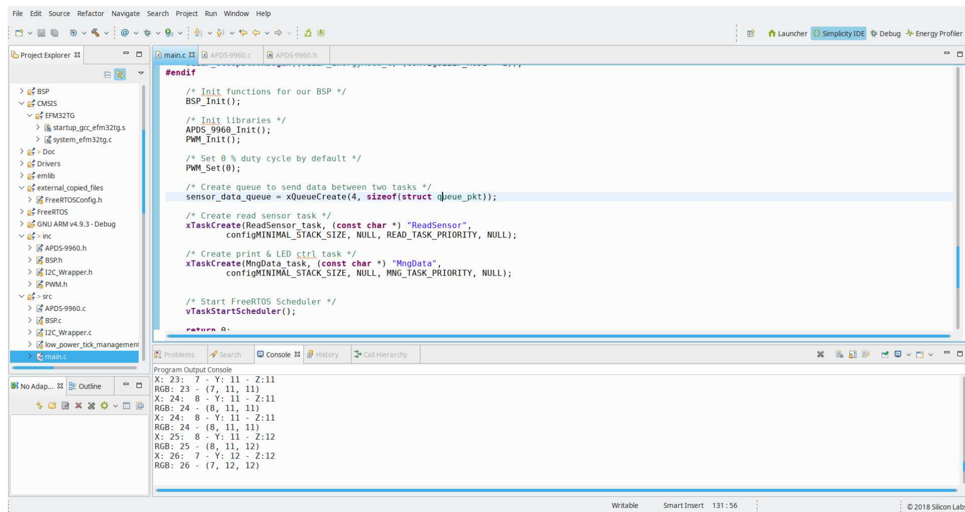


Figura 1.3: Aspecte del IDE Simplicity Studio™de SiliconLabs

Aquesta pàgina està en blanc expressament, tot va bé.

2. Breu introducció als sistemes encastats

Un sistema encastat és, bàsicament, un seguit de components hardware i software treballant conjuntament per obtenir una aplicació o funcionalitat determinada.

Els components hardware es poden dividir en tres grans blocs:

- procés de dades: dispositiu amb la capacitat de gestionar dades, entrada sortida, etc. Pot ser un microcontrolador, un DSP[1], una FPGA[2], un ASIC[3] o un sistema hibrid que incorpori varis dels anteriors dins el mateix hardware.
- sensors o introducció de dades. Qualsevol dispositiu que rep estímuls del mon físic i els converteix en dades, ja siguin digitals o analògiques (termòmetre, pantalla tàctil, acceleròmetre, etc.).
- actuadors o presentació de dades. Qualsevol dispositiu que rep una dada o sèrie de dades i ho converteix en una acció física (motor, pantalla, relé, etc.).

2.1 Microcontroladors

Un microcontrolador està compost bàsicament d'una CPU, memòria RAM i ROM, i un seguit de perifèrics tot integrat en un sol dispositiu. La varietat de microcontroladors diferents que es poden trobar al mercat és ingent, fent impossible fer un llistat de totes les opcions disponibles.

Històricament cada fabricant de microcontroladors tenia les seves pròpies famílies de dispositius (amb la seva pròpia arquitectura) amb les eines necessàries i, habitualment, cada conjunt era diferent i incompatible amb els demès. Durant els últims anys aquest panorama ha canviat força, ja que molts fabricants han acabat adoptant una sola arquitectura. Aquesta arquitectura és l'anomenada **Cortex** de l'empresa ARM [8]. Els fabricants trien quin model de CPU volen incorporar al seu dispositiu i hi afegeixen els perifèrics del seu catàleg que consideren que cal a cada dispositiu

[1] Digital Signal Processor (Processador Digital de Senyal)
[2] Field Programmable Gate Array
[3] Application Specific Integrated Circtuit (circuit integrat d'aplicació específica)

concret. D'aquesta manera, un sol fabricant té desenes o centeners de dispositius diferents basats en una mateixa CPU i amb diferents combinacions i nombre de perifèrics i memòria.

Tot i que tenim diferents dispositius de diferents fabricants que poden tenir la mateixa CPU, un executable no serà compatible entre aquests dispositius, ja que, segurament, el mapa de memòria o els diferent perifèrics seran diferents i, per tant, incompatibles.

Alguns dels fabricants més reconeguts de microcontroladors actuals són els següents:

- Silicon Labs (SiLabs) [9]
- Texas Instruments (TI) [10]
- NXP Semiconductors (NXP) [11]
- STMicroelectronics (ST) [12]
- Microchip, antiga Atmel [13]

2.2 ARM Cortex

Com ja hem dit, l'arquitectura Cortex és actualment una de les més esteses en el camp dels processadors i microcontroladors de 32 bits. Es presenten tres perfils diferents segons l'àmbit d'aplicació:

- **Cortex-A** (d'**A**plicació) Processadors d'altes prestacions per usos en telèfons mòbils, servidors, *tablets*, etc. [14].
- **Cortex-R** (de Temps **R**eal) Dissenyats per aplicacions amb alts requeriments de seguretat com dispositius mèdics, aviònica o PLCs [15].
- **Cortex-M** (de **M**icrocontrolador) CPUs dissenyats per microcontroladors i aplicacions de baix consum [16].

En aquest llibre i curs ens centrarem exclusivament a parlar del Cortex-M i les seves versions.

2.2.1 Cortex-M

Aquesta arquitectura proposada per ARM és una arquitectura tipus RISC de 32 bits amb suport de *cache*, punt flotant i dissenyada per ser de baix consum. Tot i que hi ha diferents versions d'aquesta arquitectura, mantenen un conjunt d'instruccions comuns. Aquí llistem els més habituals [17, pàgina 7]:

- **Cortex-M0+**: És la versió més bàsica d'aquesta arquitectura, orientat a dispositius molt barats i senzills i de molt baix consum. Conté un *pipeline* de 2 etapes i no té cap mena de *cache*. És la versió més senzilla que es pot trobar als catàlegs dels fabricants [18].
- **Cortex-M3**: Versió de millors prestacions, amb un *pipeline* més llarg (3 etapes) i predicció de salts, suport HW d'operacions amb punt fix, sistema de debug avançat i, opcionalment, protecció de memòria [19].
- **Cortex-M4**: Versió que afegeix capacitats de punt flotant en HW a un Cortex-M3 [20].
- **Cortex-M7**: Versió d'alt rendiment amb un *pipeline* de 6 etapes superescalar, predicció de salts i aritmètica de punt flotant de fins a 64 bits [21].

Taula 2.1: Rendiment de diferents famílies de Cortex-M [22, pàgina 22]

	M0+	M3	M4	M7
Dhrystone (DMIPX/MHz)	0.94	1.25	1.25	2.14
CoreMark/MHz	2.42	3.32	3.40	5.04

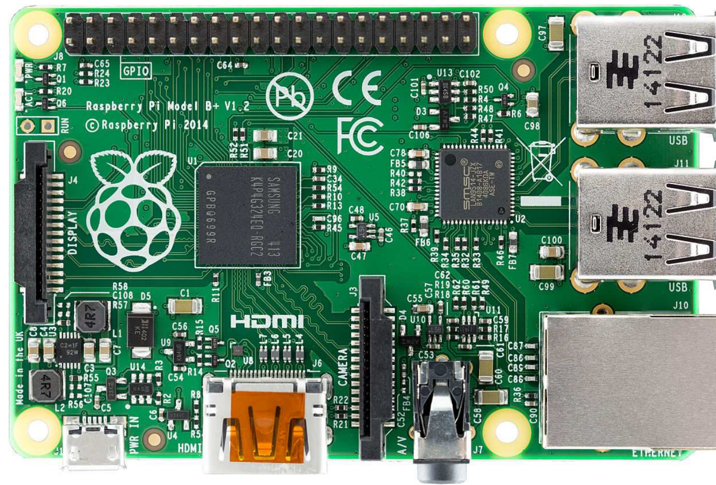


Figura 2.1: Raspberry Pi amb un processador Cortex-A
Raspberry Pi amb un processador Cortex-A. Per Lucasbosch [CC BY-SA 3.0], de la Wikimedia Commons https://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Raspberry_Pi_B%2B_top.jpg/512px-Raspberry_Pi_B%2B_top.jpg

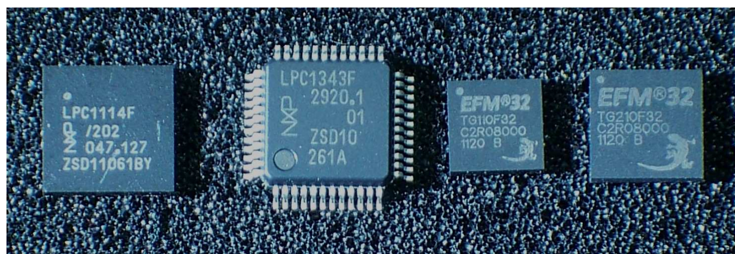


Figura 2.2: Diferents microcontrolador Cortex-M0 i M3
Diferents microcontrolador Cortex-M0 i M3. Per Viswesr [CC BY-SA 3.0], de la Wikimedia Commons https://upload.wikimedia.org/wikipedia/commons/thumb/3/3d/ARM_Cortex-M0_and_M3_ICs_in_SMD_Packages.jpg/512px-ARM_Cortex-M0_and_M3_ICs_in_SMD_Packages.jpg

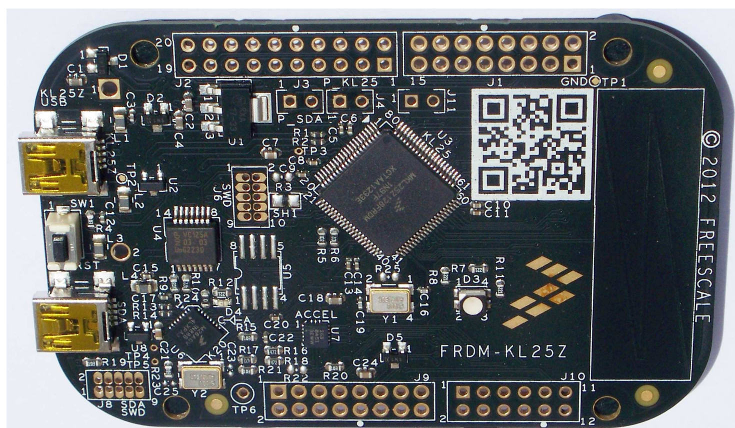


Figura 2.3: Placa Freescale FRDM-KL25Z amb un Cortex-M0+
Placa Freescale FRDM-KL25Z amb un Cortex-M0+. Per Viswesr [CC BY-SA 3.0], de la Wikimedia Commons https://commons.wikimedia.org/wiki/File:Freescale_FRDM-KL25Z_board_with_KL25Z128VLK_(ARM_Cortex-M0%2B_MCU).JPG

2.3 Arquitectura

L'arquitectura dels processadors Cortex-M varia segons la família que estudiem. La família Corte-M0+ té una arquitectura Von Newmann i la resta de famílies tenen arquitectura Harvard mixta, on tot i que la CPU té busos separats per accedir a l'espai de dades o a l'espai de codi, estan tots dos connectats a una sola matriu (Figura 2.4) [23, pàgina 794].

A part de la CPU pròpiament dita, el que se'n diu *core* en els microcontroladors ARM inclou també un controlador d'interrupcions (veure **Capítol 7 - Controlador d'interrupcions**), un *timer* simple (**Secció 5.1 - *Systick***) i un mòdul opcional de protecció de memòria (MPU) [23, pàgina 230]. Al ser dispositius comuns a tots els *Cores*, l'accés a ells és molt similar entre diferents fabricants (veure **Secció 37.1 - CMSIS-Core**).

2.3.1 Perifèrics mapats a memòria

A l'arquitectura Cortex els perifèrics estan mapats a memòria (*memory mapped*). Això fa que tinguem accessibles els registres de cada perifèric a una regió de memòria determinada. Així doncs, per accedir als registres d'un perifèric per configurar-lo o accedir a les seves dades el que caldrà fer és accedir a una certa adreça de memòria de la forma habitual i llegir o escriure la dada que pertoqui. Aquest mapa de memòria depèn de cada fabricant i model.

Per exemple, en el Cortex-M3 de la nostra placa, el mapa de memòria és el següent (resumit) (Figura 2.4):

- de 0x0000_0000 fins a 0x1FFF_FFFF: Codi, aquí hi ha mapat la Flash del microprocessador, incloent-hi la memòria de codi principal, i alguna zona tipus FLASH per l'usuari.
- de 0x2000_0000 fins 0x2000_3FFF la memòria RAM del microprocessador.
- de 0x4000_0000 fins 0x40FF_FFFF estan mapats tots els perifèrics que conformen el microcontrolador. Per exemple:
  - 0x4000_6000 fins 0x4000_6FFF hi ha el controlador de GPIO[4]
  - 0x4000_2000 fins 0x4000_2FFF hi ha l'ADC0.
  - 0x4000_6000 fins 0x4000_6FFF hi ha el controlador de GPIO
- etc.

Dins de la primera zona hi trobem la zona DI (*Device Information*) de l'adreça 0x0FE0_8000 fins a 0x0FE0_8400. Aquesta zona guarda certs valors únics per a cada dispositiu. En aquest espai, els registres MEM_INFO_FLASH, MEM_INFO_RAM i PART_FAMILY els podem llegir fàcilment [24, pàgina 24] (Figura 2.5):

Al codi del Llistat 2.1 es defineixen les 3 adreces de memòria per ser accedides fent servir un punter. Així, llegint el valor de les definicions FLASH_INFO, RAM_INFO o PART_INFO s'accedeix a la posició de memòria definida de forma directa. Per fer una escriptura es faria igual, però en l'exemple no es pot escriure a cap d'aquests registres. Debugant el codi línia a línia veurem que la variable aux pren el valor corresponent a cada un dels registres mapats.

Enlloc d'accedir a cada registre per separat, com hem fet a l'exemple, es pot definir una estructura que es correspongui amb cada un dels registres d'un perifèric en concret i que aquesta apunti a l'adreça base del perifèric. Així, per accedir a un registre en concret només caldrà accedir al camp de l'estructura definida.

Un exemple d'això el tenim a fitxer efm32g_devinfo.h, que defineix una estructura d'aquest estil, com es veu al Llistat 2.2.

[4] General Purpouse Input/Output

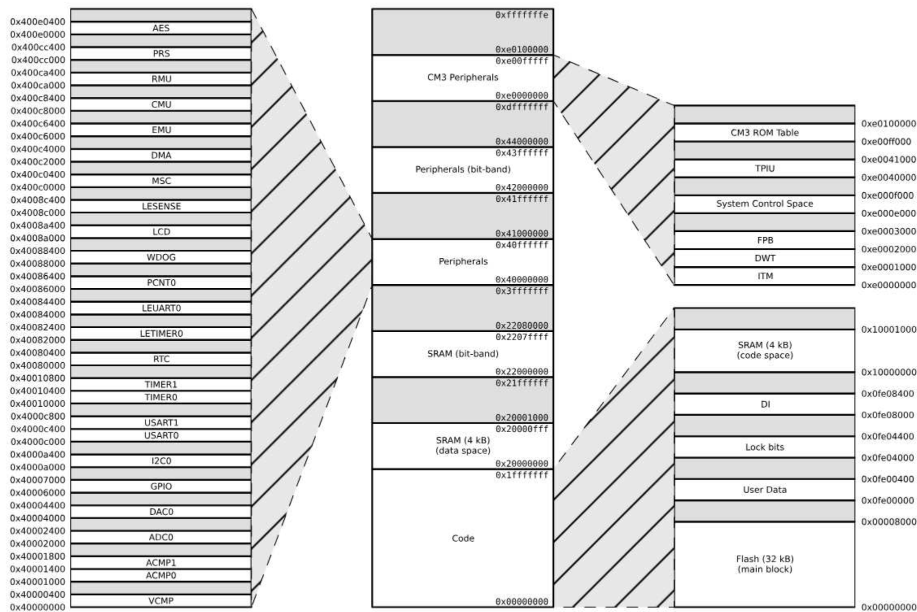


Figura 2.4: Mapa de memòria d'un Cortex-M3
Mapa de memòria d'un Cortex-M3 (EFM32TG Reference Manual [24] ©SiliconLabs)

0x0FE081F8	MEM_INFO_FLASH	[15:0]: Flash size, kbyte count as unsigned integer (eg. 128).
0x0FE081FA	MEM_INFO_RAM	[15:0]: Ram size, kbyte count as unsigned integer (eg. 16).
0x0FE081FC	PART_NUMBER	[15:0]: EFM32 part number as unsigned integer (eg. 230).
0x0FE081FE	PART_FAMILY	[7:0]: EFM32 part family number (Gecko = 71, Giant Gecko = 72, Tiny Gecko = 73, Leopard Gecko=74, Wonder Gecko=75).
0x0FE081FF	PROD_REV	[7:0]: EFM32 Production ID

Figura 2.5: Registres de la DI a usar a l'exemple [24, pàgina 24]

Llistat 2.1: Accedint a memòria en C

```c
#define FLASH_INFO   (*(unsigned char *)0x0FE081F8)
#define RAM_INFO     (*(unsigned char *)0x0FE081FA)
#define PART_INFO    (*(unsigned char *)0x0FE081FE)

volatile uint32_t aux;

int main(void) {
  while (1) {
    aux = FLASH_INFO; /* 32 kB */
    aux = RAM_INFO;   /* 4 kB */
    aux = PART_INFO;  /* 73 = Tiny Gecko */
  }
}
```

Llistat 2.2: Exemple de definició d'estructura per accedir a memòria

```
typedef struct
{
  __IM uint32_t CAL;            /**< Calibration temperature and checksum */
  __IM uint32_t ADC0CAL0;       /**< ADC0 Calibration register 0 */
  __IM uint32_t ADC0CAL1;       /**< ADC0 Calibration register 1 */
  __IM uint32_t ADC0CAL2;       /**< ADC0 Calibration register 2 */
  uint32_t RESERVED0[2];        /**< Reserved */
  __IM uint32_t DAC0CAL0;       /**< DAC calibrartion register 0 */
  __IM uint32_t DAC0CAL1;       /**< DAC calibrartion register 1 */
  __IM uint32_t DAC0CAL2;       /**< DAC calibrartion register 2 */
  __IM uint32_t AUXHFRCOCAL0;   /**< AUXHFRCO calibration register 0 */
  __IM uint32_t AUXHFRCOCAL1;   /**< AUXHFRCO calibration register 1 */
  __IM uint32_t HFRCOCAL0;      /**< HFRCO calibration register 0 */
  __IM uint32_t HFRCOCAL1;      /**< HFRCO calibration register 1 */
  __IM uint32_t MEMINFO;        /**< Memory information */
  uint32_t RESERVED2[2];        /**< Reserved */
  __IM uint32_t UNIQUEL;        /**< Low 32 bits of device unique number */
  __IM uint32_t UNIQUEH;        /**< High 32 bits of device unique number */
  __IM uint32_t MSIZE;          /**< Flash and SRAM Memory size in KiloBytes */
  __IM uint32_t PART;           /**< Part description */
} DEVINFO_TypeDef; /** @} */
```

Llistat 2.3: Declaració d'una variable d'accés a la memòria estructurada

```
#define DEVINFO ((DEVINFO_TypeDef *) DEVINFO_BASE) /**< DEVINFO base ptr */
```

Que es correspon amb els registres de la regió DI a la que hem accedit abans. Aquesta estructura es fa servir al fitxer efm32tg840f32.h definint la referència mostrada al Llistat 2.3.

De manera que es pot accedir als mateixos registres com s'indica al Llistat 2.4. Que és una forma bastant més còmoda de treballar.

Per sort, la majoria de fabricants proporcionen llibreries de baix nivell que ens estalvien tant conèixer tots els detalls de cada un dels perifèrics com d'haver de manegar els registres un a un: pel cas de SiliconLabs aquestes llibreries s'agrupen sota la EMLIB [25]; en el cas de l'empresa ST ens proporciona la biblioteca *STM32 Standard Peripheral Libraries* [26] o la més moderna *STM32Cube hardware abstraction layer (HAL)* [27].

El codi d'aquests exemples està al repositori del curs

2.3.2 Mida del codi i seccions de memòria

Quan compilem el nostre codi, com ja sabem, el compilador trasllada el codi C i assemblador a codi màquina, generant un fitxer per cada mòdul que formi l'aplicació (anomenats fitxers objecte amb

Llistat 2.4: Ús de l'estructura d'accés

```
aux = DEVINFO->MSIZE;
aux = DEVINFO->PART;
```

extensió .o). A continuació el *linker* agafa tots els fitxers objecte i crea el fitxer binari tipus ELF.

En aquest fitxer hi ha tota la informació necessària per programar el microcontrolador, i això inclou el codi màquina que cal executar, les definicions de les variables i la seva inicialització, el codi d'inicialització (veure **Subsecció 2.3.3 - Procés de *boot***), etc. Aquest fitxer serà el que després el programador o *debugger* llegirà per tal de programar el microcontrolador.

D'aquest fitxer podem extreure informació valuosa, com és la quantitat de memòria FLASH o RAM que necessita el nostre programa. Això ens ho diu la comanda *size* (en el cas del compilador GCC per ARM la comanda és *arm-none-eabi-size*) quan s'executa sobre el fitxer binari creat. La sortida d'aquest programa té el següent aspecte:

```
text    data    bss    dec    hex  filename
4976     112     48   5136   1410  SpeedTest_1.axf
```

El que està mostrant és la mida (en bytes) de cada un dels segments en que es divideix la nostra aplicació:

- **text**: aquest sector és el corresponent al codi executable i les constants definides; també s'inclouen aquí els vectors d'interrupció (veure **Capítol 7 - Controlador d'interrupcions**). El *debugger* s'encarregarà de gravar a la FLASH aquesta secció.
- **data**: s'emmagatzemem les dades inicialitzades, com són variables han d'anar a RAM, però també cal guardar el seu valor a la FLASH, per tant, ocupen espai a totes dues memòries. El procés de *boot* (Veure **Subsecció 2.3.3 - Procés de *boot***) copiarà els valors d'inicialització de la FLASH cap a la variable a la RAM.
- **bss**: conté totes les variables no inicialitzades. Aquesta secció va a la RAM. Al procés de *boot* (Veure **Subsecció 2.3.3 - Procés de *boot***) aquesta secció s'inicialitza amb zeros.
- **dec**: és la suma dels 3 camps anteriors
- **hex**: és el mateix valor que **dec** però expressat en hexadecimal.

2.3.3 Procés de *boot*

El procés de *boot* és tota la seqüència de passes que fa un microcontrolador des de que s'engega fins que comença a executar la nostra funció principal **main()**.

Quan el microcontrolador surt de l'estat de *reset* després d'un *power-up*, d'un *reset* extern, d'un *reset* pel Watchdog (veure **Capítol 11 - *Watchdog***), etc. cal que s'inicialitzin un seguit de mòduls i peces abans no es pugui començar a executar el nostre codi.

A la posició de memòria 0x0000_0000 el que hi ha és el vector d'interrupció corresponent al *reset* que normalment crida la funció **SystemInit()** que inicialitza, si cal, parts crítiques del microcontrolador, com el rellotge principal. A continuació es copien les variables inicialitzades de la FLASH a la RAM (secció **data**), s'inicialitza a 0 la secció coneguda com **bss** i per últim crida a la funció **_start()** que acabarà cridant a la nostra funció **main()** [28]. Tot això ho maneguen les eines de forma automàtica i no cal que nosaltres en tinguem cura, però va bé saber què està passant dins el microcontrolador en tot moment.

2.4 Rapidesa d'un microcontrolador

Molts cops no ens fem a la idea de com de ràpid és la CPU d'un microcontrolador. Estem acostumats a llegir i escoltar freqüències de funcionament dels microprocessadors d'escriptori o de servidor, que actualment són de Gigahertz i els nostres pobres microcontroladors van, en el millor dels casos, a uns quants pocs Megahertz. Això ens pot fer pensar que els nostres microcontroladors són lents i que no poden fer gaire coses.

Llistat 2.5: Codi velocitat d'un microcontrolador

```
void main() {
  ...
  /* while button 0 is pressed, CPU is counting */
  while (GPIO_PinInGet(gpioPortD, 8) == 0) {
    i++;
  }

  if (i != 0) {
    printf("i = %d\n", i); i = 0;
  }
  ...
}
```

Taula 2.2: Mesures de temps i sumes per segon

Count	Ticks (del Timer)	Temps (segons)	Count/Tick	Count/Segon
16629	1688	0,12	9,85	134.677
10955	1112	0,08	9,85	134.681
81813	8309	0,61	9,85	134.609
17985	1826	0,13	9,85	134.651
12054	1224	0,09	9,85	134.653
269785	27400	2,00	9,85	134.607

Podem comprovar-ho empíricament.

A l'exemple SpeedTest_1 hi ha un codi molt simple que simplement incrementa un comptador mentre es té premut el botó 0 i tot seguit imprimeix per la consola el valor al que ha arribat el comptador.

Això ens hauria de donar una idea de com de ràpid és capaç la nostra CPU de fer operacions aritmètiques simples.

Pitjant el botó molt ràpidament el comptador arriba a 8.000 i 9.000. Així que sembla que compta molt ràpid!

2.4.1 Millor mesura de temps

El segon exemple és una mica més complicat. Per tal de mesurar el temps que es pitja el botó, farem servir un *Timer*, que ja ho veurem més endavant (veure **Capítol 8 - Timers**). El que es fa a l'exemple és mesurar acuradament el temps que està el botó pitjat i calcular el nombre d'operacions que s'ha fet en aquell temps.

Com podem veure a la Taula 2.2 el nombre d'operacions per segon es manté constant independentment del temps que estiguem pitjant el botó i és un número força alt, 134.000 sumes per segon!!!

II Programació de perifèrics I

3 Consola de Debug **31**

4 Fent servir printf **33**
4.1 Problemes d'usar printf

5 Gestió de rellotges **37**
5.1 *Systick*

6 GPIO **41**
6.1 Un exemple senzill
6.2 BSP
6.3 Manipulant bits individuals

7 Controlador d'interrupcions **47**
7.1 Escrivint ISRs en C
7.2 Fent servir ISRs

8 Timers **51**
8.1 Exemple senzill amb un Timer
8.2 Exemple més complex amb el Timer

9 RTC **57**
9.1 RTC externs

10 PWM **61**
10.1 Generar PWM
10.2 Controlant un servomotor

11 *Watchdog* **69**
11.1 Exemple

Aquesta pàgina està en blanc expressament, tot va bé.

En aquest capítol s'aniran introduint i explicant els perifèrics mes habituals que trobem en els microcontroladors actuals. Cada capítol constarà d'una petita introducció al perifèric en qüestió i un petit exemple amb el codi corresponent i els comentaris adients.

R El tipus de codi que es veurà als exemples es coneix com *Baremetal* que vol dir que no es fa servir cap Sistema Operatiu, que es veurà a la **Part IV - FreeRTOS**. Aquests sistemes es basen en les inicialitzacions necessàries i un bucle sense fi al **main()** on es realitzen les operacions desitjades. S'acostuma a usar aquest mètode en aplicacions senzilles o en aplicacions que corren en microcontroladors poc potents o sense prou memòria com per executar còmodament un RTOS[5].

[5] *Real-Time Operating System*, Sistema Operatiu de Temps Real

Aquesta pàgina està en blanc expressament, tot va bé.

3. Consola de Debug

Un de les principals diferències quan treballem amb sistemes encastats és que no tenim una consola on executem el nostre binari i podem veure quins resultats ha obtingut.

Una millora d'ARM respecte arquitectures anteriors va ser la d'incorporar ja fa temps un mecanisme de debug, a través d'un pin d'output anomenat *SWO* que permet enviar dades cap a una consola al nostre PC de desenvolupament.

Aquest pin *SWO* forma part del sistema de Debug dels Cortex anomenat ITM (*Instrumentation Trace Macrocell*) [29]. Això vol dir que la majoria de microcontroladors basats en Cortex que ens trobem, siguin del fabricant que sigui portaran aquesta funcionalitat.[1]

Per poder fer servir aquesta funcionalitat, cal primer configurar el pin SWO per a que funcioni com a tal (es pot fer servir també com un GPIO normal). Tot seguit es configura el dispositiu de trace i el mòdul ITM d'aquest. La configuració es fa a la funció **setupSWOForPrint()**.

Un cop configurat el mòdul ITM, la funció **ITM_SendChar()** permet enviar caràcter a caràcter el que vulguem presentar a la consola a l'altre costat.

En el cas de Simplicity, els caràcters rebuts es presenten directament a la consola de la part d'abaix del IDE (Figura 3.1).

[1] La majoria de Cortex-M0+ no porten aquest perifèric.

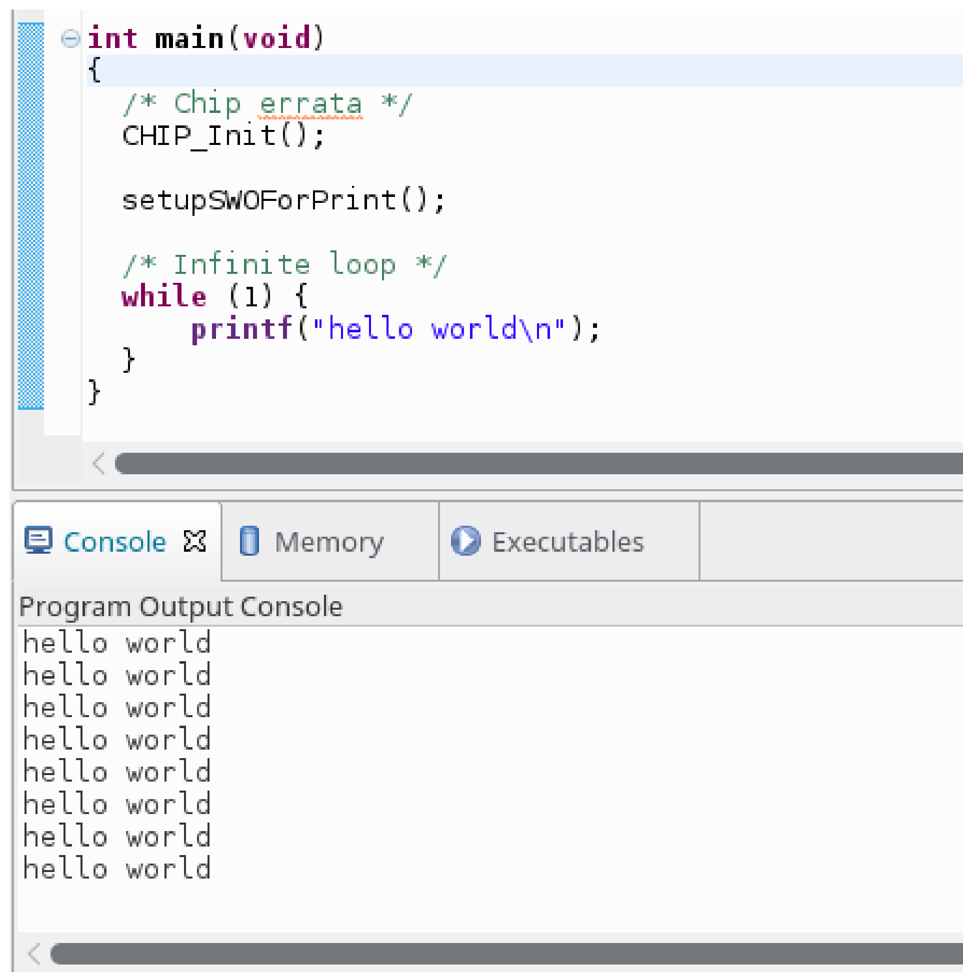


Figura 3.1: Captura de pantalla de la Consola del Simplicity Studio

4. Fent servir printf

Tot i que això és força útil, podem fer servir una funció força coneguda i molt útil com és **printf()** per fer-nos-ho tot més senzill (a canvi d'alguna cosa, com veurem).

La funció **printf()** és una vella coneguda per qualsevol programador de C (o C++, o PHP, o ...). Així que seria genial poder fer servir aquesta funció en el nostre sistema encastat i que la cadena aparegui a la consola del nostre PC. Això ho podem fer de la següent manera. Primer de tot cal saber que la funció **printf()** fa servir la funció de sistema **_write()** per imprimir la cadena. Per tant, caldrà que implementem aquesta funció per tenir el nostre desitjat **printf()**.

Com podem veure al Llistat 4.1, el que fa aquesta funció és anar enviant un a un tots els caràcters de la cadena passada via el paràmetre **ptr** i de longitud **len**.

També cal configurar el mòdul ITM (mòdul de debug) perquè activi la sortida *SWO* cridant a la funció **setupSWOForPrint()**. Un cop fet això, ja podem fer servir la funció **printf()** tal com hem fet sempre.

Aquesta explicació la podeu trobar al fòrum de Silicon Labs (no cal registre) i el codi el teniu en el directori d'instal·lació del Simplicity/developer/sdks/exx32/v4.4.1/kits/common/drivers/ als fitxers:

- retargetio.c

Llistat 4.1: Funció **_write()**

```
int _write(int file, const char *ptr, int len) {
  int x;
  for (x = 0; x < len; x++) {
    ITM_SendChar(*ptr++);
  }
  return (len);
}
```

- retargetswo.c

4.1 Problemes d'usar printf

Com dèiem, poder fer servir el nostre estimat **printf()** al nostre sistema encastat no ens sortirà gratis. Com que aquesta funció és força complexa i permet moltes possibilitats, incloure-la en un projecte afegirà una bona quantitat de memòria de programa.

En l'exemple que tenim al repositori, les mides són les que es veuen a la Taula 4.1. Per tant, podem estimar que afegir **printf()** al nostre projecte afegirà uns 3800 Bytes (3.7 KB) de codi de programa.

Taula 4.1: Mida de l'executable segons **printf()**

Opció	Bytes secció .text
Sense printf()	956
Amb printf()	4.748
Amb printf() i punt flotant	13.644

Potser no és gaire important aquesta quantitat, però segur que caldrà tenir-la en compte si estem treballant amb microcontroladors que tenen poca memòria FLASH de programa (hi ha Cortex-M0+ amb només 4 KB de FLASH!).

També cal tenir en compte que algunes versions de la funció **printf()** no suporten valors en punt flotant. Segons l'eina, caldrà activar aquesta opció en cas que la vulguem fer servir (Figura 4.1. Cal tenir en compte que això incrementarà encara més la quantitat de memòria que necessitarà aquesta funció com es veu a la Taula 4.1.

Una forma força habitual de disposar dels avantatges del printf mentre es desenvolupa i treure'l de forma ràpida quan es genera el binari definitiu es redefinir el **printf** amb un nom nostre, i establir una variable condicional de compilació per activar o no el printf real, tal i com es veu en el llistat Llistat 4.2, llavors en el nostre codi, enlloc de cridar **printf** per mostrar un missatge, haurem de fer servir **PRINTF**.

Llistat 4.2: Redefenir **printf()**

```
#ifdef USE_PRINTF
  #define PRINTF(...) printf( __VA_ARGS__)
#else
  #define PRINTF(...)
#endif
```

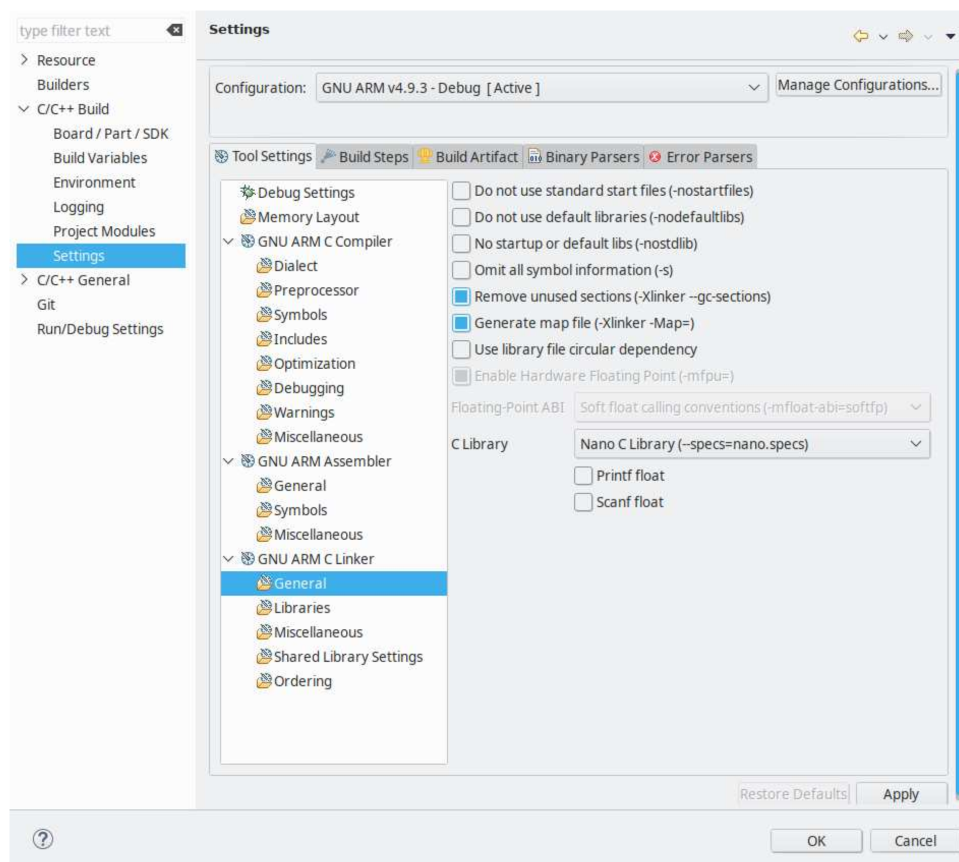


Figura 4.1: Opció a *Simplicity* per activar el punt flotant al **printf()**

Aquesta pàgina està en blanc expressament, tot va bé.

5. Gestió de rellotges

En la majoria de microcontroladors moderns la gestió dels rellotges és una qüestió delicada i molt important. Per tal de millorar el consum del dispositiu, és habitual tenir un control i poder decidir si cert perifèric rep el senyal de rellotge o no (més detalls a **Capítol 35 - Baix cosum**). En cas que no el rebi, el perifèric romandrà totalment desconnectat i no consumirà energia. En cas que el vulguem fer servir, una de les primeres coses que haurem de fer és activar i proporcionar-li el senyal de rellotge adequat.

Per aquesta tasca de controlar els rellotges, acostuma a existir un perifèric concret que fa tota la gestió, tant de manegar l'entrada de diferents senyals de rellotge com de preparar i enviar aquests senyals als diferents perifèrics. En els microcontroladors tant de SiliconLabs com de ST tenim diferents branques de rellotge per diferents perifèrics i la CPU. En termes generals podem dir que els perifèrics considerats lents (RTC, LEUART, etc.) reben un senyal de rellotge de baixa freqüència, els perifèrics considerats ràpids (USART, SPI, DAC, ADC, Timers, etc.) un senyal de rellotge d'alta freqüència i la CPU i els perifèrics més relacionats (DMA, Interrupcions, etc.) un altre rellotge [24, pàgina 94][30, pàgina 152].

Al llarg dels diferents exemples s'anirà veient com es gestionen els rellotges. En els casos més senzills, tant sols cal activar el rellotge pel perifèric desitjat cridant a la funció **CMU_ClockEnable()**. Aquesta funció rep com a paràmetre el perifèric al que se li vol enviar o desactivar el rellotge.

Altres funcions permeten decidir quin senyal rellotge concret es connecta amb quina branca (funció **CMU_ClockSelectSet()**); dividir un rellotge abans d'entrar a cert perifèric (**CMU_ClockDivSet()**) (Llistat 5.1).

Llistat 5.1: Exemple de configuració del rellotge pel RTC

```
CMU_ClockSelectSet( cmuClock_LFA, cmuSelect_LFXO ); // El rellotge "Low
    Frequency Crystal Oscillator" entra al bus LFA
CMU_ClockDivSet( cmuClock_RTC, cmuClkDiv_32768 );  // El rellotge es divideix
    per 32768 abans d'alimentar el RTC
CMU_ClockEnable(cmuClock_RTC, true);   // S'activa el rellotge pel periferic
    RTC
CMU_ClockEnable(cmuClock_HFLE, true );  // S'activa el rellotge "Low energy
    clock"
```

5.1 *Systick*

Com ja s'ha mencionat breument a **Secció 2.3 - Arquitectura**, dins els *cores* ARM-M hi ha un *timer* simple sense cap relació amb els *Timers* perifèrics que veurem més endavant a **Capítol 8 - Timers**. Aquest *timer* es coneix pel nom de *Systick* i consta tan sols d'un comptador decreixent de 24 bits i un generador d'interrupció en quant arriba a zero [23, pàgina 312]. El timer Systick funciona amb el rellotge de la CPU dividit per algun factor configurable. Això caldrà tenir-ho en compte alhora de fer implementacions de baix consum, on en ocasions s'atura aquest rellotge (veure **Capítol 35 - Baix cosum**).

Aquest *timer* està pensat perquè els S.O. el puguin fer servir, i com que està integrat dins el *core* Cortex, la portabilitat dels S.O. entre diferents fabricants serà molt senzilla (seria més complicat haver de fer un port per cada *timer* diferent de cada fabricant).

L'exemple al repositori implementa una funció de **Delay()**. El que es fa és configurar el *Systick* perquè generi una interrupció cada 1 mil·lisegon (Llistat 5.2). Com que la funció **CMU_ClockFreqGet (cmuClock_CORE)** retorna la freqüència de funcionament del rellotge del sistema, al dividir-la per 1000 i configurar el Systick amb aquest valor, generarà una interrupció cada mil·lisegon.

A la ISR corresponent s'incrementa una variable global per tenir un comptatge dels mil·lisegons transcorreguts (veure Llistat 5.3). La variable **msTicks** s'ha definit com a **volatile** pel que s'explicarà a **Subsecció 7.2.1 - Ús de variables globals**.

Per últim, la funció **Delay()** rep com a paràmetre els mil·lisegons a aturar-se i s'espera aquest temps comptant el temps fent servir la variable global que incrementa la ISR (Llistat 5.4).

Llistat 5.2: Configuració del *Systick*

```
main() {
  ...
  SysTick_Config(CMU_ClockFreqGet(cmuClock_CORE) / 1000);
  ...
}
```

Llistat 5.3: ISR del *Systick*

```
void SysTick_Handler(void)
{
  msTicks++;        /* increment counter necessary in Delay()*/
}
```

Llistat 5.4: Funció delay() amb *Systick*

```
void Delay(uint32_t dlyTicks)
{
  uint32_t curTicks;

  curTicks = msTicks;
  while ((msTicks - curTicks) < dlyTicks) ;
}
```

Aquesta pàgina està en blanc expressament, tot va bé.

6. GPIO

Diem GPIO[1] al perifèric encarregat de la gestió de l'entrada i sortida de propòsit general. Fent servir aquest perifèric podrem configurar l'entrada o la sortida d'un pin concret del microcontrolador i posar-hi el valor desitjat ('0' o '1') o llegir quin valor hi ha posat algun altre dispositiu.

De forma general, un pin en concret el podrem configurar perquè treballi com a entrada o com a sortida. Si un pin està configurat com a entrada, el valor de voltatge elèctric que tingui a l'entrada del pin, es podrà llegir per part del codi del microcontrolador. De forma inversa, un pin configurat com a sortida posarà el valor elèctric equivalent al valor que el codi escrigui.

Així, si estem treballant a 3.3 Volts d'alimentació i d'entrada i sortida, si a un pin configurat com d'entrada algun altre dispositiu hi posa un valor proper a 3.3 volts, des de el nostre software o comunament anomenat **Firmware** (FW) en la seva forma anglesa, llegirem que aquest pin té un valor d''1'. En canvi, si el pin està configurat com a sortida, quan posem un '1' des del FW, el pin corresponent forçarà un valor de 3.3 volts (com a la Figura 6.1).

R Quan diem que l'alimentació és de 3.3 Volts estem suposant aquesta tensió d'alimentació, però el rang acceptable va de 1.8 fins a 3.8 Volts i llavors les sortides tindrien el valor d'alimentació [31, pàgina 9]. Així, si alimentem el microcontrolador a, posem per cas, 2.8 Volts, un '1' lògic de sortida d'un pin forçarà 2.8 Volts a aquell pin.

Existeixen diferents formes de configurar un pin segons el fabricant i la tecnologia, la més comuna és el mode *push-pull* que permet forçar un valor '1' o '0' segons convingui. Una altra opció que de vegades cal fer servir és el mode *open-drain*, que la sortida només pot forçar el valor '0' però no el valor '1'.

En aquest cas, per forçar el valor '1' es fa servir una resistència connectada a 3.3 volts; aquesta mena de resistència s'anomena un *pull-up*. Aquesta mena de resistències (o el seu complementari,

[1] General Purpouse Input/Output

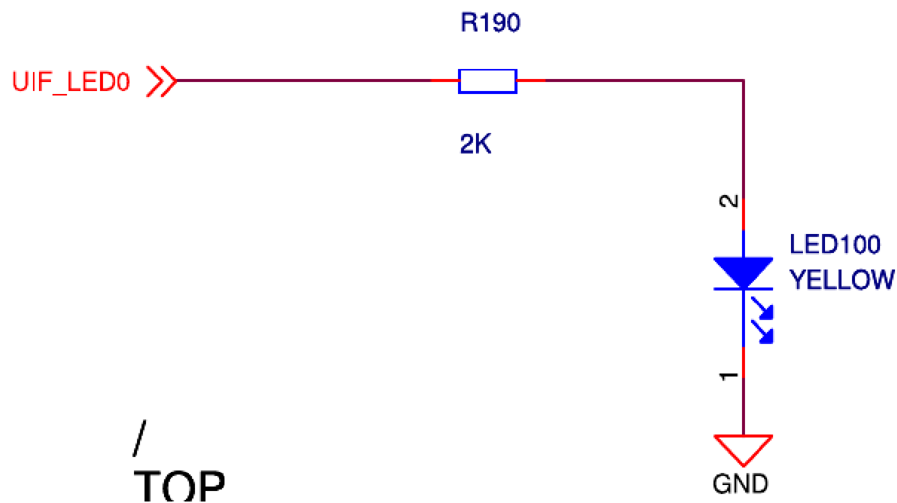


Figura 6.1: Esquemàtic mostrant un LED connectat a un pin de GPIO

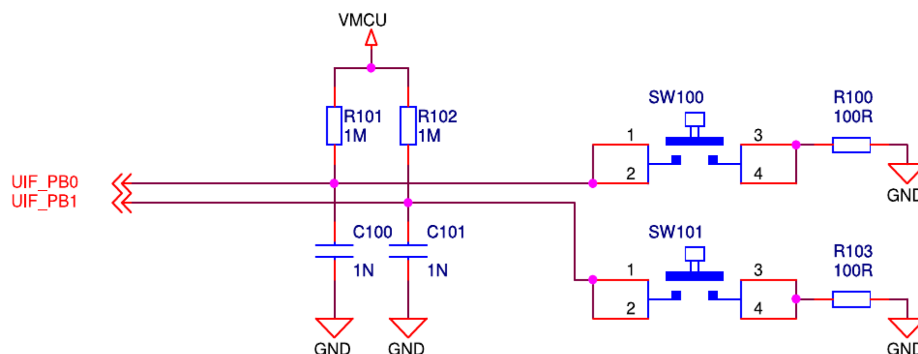


Figura 6.2: Esquemàtic amb resistències de *pull-ups* (etiquetades com R101 i R102)

un *pull-down*) també s'utilitza en la connexió de polsadors o botons, de manera que quan el botó no està polsat, la resistència de *pull-up* (o de *pull-down*) força el valor corresponent.

Com es veu al esquemàtic de la Figura 6.2, quan no es prem el botó les resistències etiquetades R101 i R102 fan que la línia estigui a '1' lògic (*pull-up*). En quan es prem el botó, aquest connecta GND a la línia i per tant passa a tenir el valor lògic '0'.

Cal saber també que la quantitat de corrent que un pin individual pot proporcionar està limitada i, en alguns casos, es pot seleccionar la quantitat màxima de corrent que pot donar un pin concret.

6.1 Un exemple senzill

Farem un codi que llegeixi l'estat dels botons de la placa, i en cas que estiguin pitjats, s'encén o apaga el LED. Veiem el codi el llistat 6.1.

El primer que es fa amb la sentència **CMU_ClockEnable()** és alimentar amb un rellotge al perifèric GPIO. En la arquitectura Cortex-M de Silicon Labs cal fer això per cada perifèric que fem servir. D'aquesta manera, un perifèric que no necessitem no rep cap rellotge i el seu consum és disminueix dràsticament.

Tot seguit es configuren els 3 pins que utilitzarem:

Llistat 6.1: Codi d'exemple de GPIO

```
CMU_ClockEnable(cmuClock_GPIO, true);

GPIO_PinModeSet(gpioPortD,  7, gpioModePushPullDrive, 0); /* LED */
GPIO_PinModeSet(gpioPortD,  8, gpioModeInput, 0); /* Boto 0 */
GPIO_PinModeSet(gpioPortB, 11, gpioModeInput, 0); /* Boto 1 */

/* Infinite loop */
while (1) {
  if (GPIO_PinInGet(gpioPortD, 8) == 0) {
    GPIO_PinOutClear(gpioPortD, 7);
  }
  if (GPIO_PinInGet(gpioPortB, 11) == 0) {
    GPIO_PinOutSet(gpioPortD, 7);
  }
}
```

- PD7 com sortida per controlar el LED,
- PD8 i PB11 com entrades connectades als botons 0 i 1.

Un cop configurats els pins, dins el bucle infinit es va mirant tota l'estona per *polling* el valor dels dos pins d'entrada i canviant el valor de sortida cap al LED segons toqui.

6.2 BSP

Podem començar a introduir el concepte de BSP[2] que no són més que funcions específiques per la nostra PCB de manera que ens aïllen la implementació de la funcionalitat.

Anem a suposar que canviem de PCB (o de versió) i el LED que volem encendre ja no està connectat al pin D7 si no que està, posem per cas, al E2. Caldria canviar totes les crides que tinguéssim al nostre codi de l'estil vist al Llistat 6.2 per la del Llistat 6.3 amb tots els errors que això provocar.

Una forma molt habitual de treballar és escriure funcions amb les funcionalitats més comunes i que ens amaguin aquests detalls. Així, pel nostre exemple podríem definir les funcions del Llistat 6.4.

Fent servir aquestes funcions enlloc de les crides directes al GPIO ens permetran introduir canvis a la PCB sense haver de canviar gaire el nostre codi.

Habitualment en el BSP s'inclouen les inicialitzacions dels diversos rellotges (veure **Capítol 5 - Gestió de rellotges**), les funcions per accedir a recursos propis de la placa com LEDs o botons, configuració de les opcions de *Debug* (Veure **Capítol 3 - Consola de Debug**), etc.

[2] *Board Support Package*

Llistat 6.2: Codi de configuració d'un pin

```
GPIO_PinOutSet(gpioPortD, 7);
```

Llistat 6.3: Codi amb la nova configuració del pin

```
GPIO_PinOutSet(gpioPortE, 2);
```

Llistat 6.4: Exemple de BSP senzill

```
LedInit() {
  GPIO_PinModeSet(gpioPortD, 7, gpioModePushPullDrive, 0); /* LED */
}

LedOn() {
  GPIO_PinOutSet(gpioPortD, 7);
}

LedOff() {
  GPIO_PinOutClear(gpioPortD, 7);
}
```

6.3 Manipulant bits individuals

Tot i que no és específic dels GPIOs, veurem aquí com manipular bits individuals en C. Sovint ens caldrà posar a valor '0' o '1' un bit individual d'una variable sense canviar el valor de la resta dels bits, veiem aquí les receptes per fer-ho.

R La numeració de bits en C comença pel 0, així el bit menys significatiu d'una variable serà sempre el 0 i el més significatiu $N-1$, amb N el nombre de bits de la variable. Per situar un 1 a un bit determinat, en C es fa servir l'operador **<<**, que desplaça cap a l'esquerra el valor de l'esquerra tants bits com indiqui el valor de la dreta.

L'exemple per aquesta secció es troba al repositori en el projecte Bit_1.

6.3.1 Posar a 1 un bit

Per posar un bit concret a '1' d'una variable tant sols cal fer una OR lògica (símbol **|**) tal com es veu al Llistat 6.5.

En aquest cas, primer es posa a '1' el bit 4 de la variable *my_variable*.

6.3.2 Posar a 0 un bit

Per posar a zero un bit individual, la feina a fer és una mica més estranya, ja que cal fer una AND lògica amb tots els bits a '1' menys el desitjat que haurà d'estar a '0'. Això es pot fer amb la mateixa construcció d'abans i fent l'operació NOT bit a bit (amb el símbol **~**) abans de fer la AND (símbol **&**).

En el mateix exemple es veu com, després de posar a 1 el bit 4, es posa a 0 el bit 3 amb la operació AND comentada.

6.3.3 *Toggle* un bit

Per a fer un *toggle* d'un bit individual, el que cal fer és la operació lògica XOR (símbol **^**) amb el bit desitjat al valor '1'.

Llistat 6.5: Manipulant un bit concret d'una variable

```
void main(void) {
  ...
  uint8_t my_variable;
  ...
  my_variable = 5;
  // Set to '1' bit 4
  my_variable |= (1 << 4);
  printf("my_variable: 0x%02X\r\n", my_variable); // should be 0x15
  // Now set to '0' bit 2
  my_variable &= ~(1 << 2);
  printf("my_variable: 0x%02X\r\n", my_variable); // should be 0x11
  // Now we toggle bit 0 twice
  my_variable ^= (1 << 0);
  printf("my_variable: 0x%02X\r\n", my_variable); // should be 0x10
  my_variable ^= (1 << 0);
  printf("my_variable: 0x%02X\r\n", my_variable); // should be 0x11
  ...

  if ( (my_variable & 0x10) != 0 ) {
    /* the variable my_variable has the 4th bit set */
  }
}
```

A l'exemple es fa *toggle* dues vegades al bit 0 de la variable.

6.3.4 Comparar si un bit està a cert valor

L'altre necessitat que apareix sovint és la de comprovar el valor d'un bit determinat d'una variable.

L'opció més habitual és fer una AND lògica entre la variable i el bit interessant i comprovar que el resultat és diferent de '0'. Si la comparació dona cert vol dir que el bit en qüestió està a '1', en cas contrari, el bit d'interès té el valor '0'. Es pot veure al final de l'exemple al Llistat 6.5, on es comprova que el bit 4 de la variable sigui valgui '1'.

R Aquesta mena d'operacions són força propicies per introduir *bugs* complicats de detectar. Operar per error un bit que no pertany a aquella mida de variable pot portar a errors molt difícils de detectar i el compilador no donarà cap mena de error o avís.

Aquesta pàgina està en blanc expressament, tot va bé.

7. Controlador d'interrupcions

Una interrupció (IRQ[1]) és un succés que interromp l'execució normal del processador i passa a executar un codi de programa especial pel succés concret. La ISR[2] és el codi que es crida per a cada succés o interrupció.

Cada interrupció té assignada una ISR pròpia. Aquesta informació s'acostuma a guardar en una zona de memòria especial, anomenada memòria de vectors d'interrupció.

Les IRQs estan enumerades i tenen prioritats, així habitualment, un valor menor vol dir major prioritat. Aquest valor de IRQ també es fa servir per saber quina posició dels vectors d'interrupció es troba la ISR corresponent.

El controlador d'interrupcions gestiona quines interrupcions rebudes arriben al processador, segons les prioritats i si la interrupció concreta està activada o no.

Veurem un cas amb els GPIO, el codi està disponible al repositori. En aquest cas, el que es fa primer és configurar els pins perquè generin una interrupció HW al flanc de baixada (recordem el *pull-up* a la PCB, Figura 6.2). Tot seguit s'activen les interrupcions corresponents.

En el cas dels Cortex-M de SiliconLabs, els pins de GPIO poden generar només 2 interrupcions, els

[1] *Interrupt request*, Petició d'interrupció
[2] *Interrupt Service Routine*, Rutina de servei d'interrupció

```
/* Set Interrupt configuration for both buttons */
GPIO_IntConfig(gpioPortD, 8, false, true, true);
GPIO_IntConfig(gpioPortB, 11, false, true, true);

/* Enable interrupts */
NVIC_EnableIRQ(GPIO_EVEN_IRQn);
NVIC_EnableIRQ(GPIO_ODD_IRQn);
```

Llistat 7.1: Exemple d'ISR per GPIO

```
void GPIO_EVEN_IRQHandler(void) {
  uint32_t aux;

  aux = GPIO_IntGet();

  /* clear flags */
  GPIO_IntClear(aux);

  /* Set LED off */
  GPIO_PinOutClear(gpioPortD, 7);
}
```

pins parells la interrupció GPIO_EVEN_IRQ i els pins senars la GPIO_ODD_IRQ [24, pàgina 405]. En els microcontroladors de ST, hi ha una arquitectura diferent i cada pin d'entrada pot generar una IRQ segons el seu índex de manera que el pin PA2, el PB2, el PC2 etc. generen la IRQ EXTI2, però només un d'aquests pins pot generar la IRQ [30, pàgina 384].

En el cos de l'exemple, com que els botons estan connectats al pin D8 i B11 cada un d'ells activarà una de les dues interrupcions.

La ISR per la interrupció parell la veiem al Llistat 7.1.

A l'arquitectura Cortex-M el nom de les ISR està fixat en un fitxer de l'entorn de programació, de manera que només cal escriure una funció amb el nom correcte i ja tenim definida la ISR. El fitxer que defineix les ISR depèn de cada model de microcontrolador, en el nostre cas és el fitxer startup_efm32tg.S.

En el cas de l'arquitectura Cortex, la pròpia ISR ha de netejar el *flag* d'interrupció que l'ha cridat. Això es fa al principi de tot de la ISR, llegint quins *flags* estan actius (funció **GPIO_IntGet()**) i tot seguit netejant aquests mateixos *flags* (**GPIO_IntClear()**).

A continuació, s'encén o s'apaga el LED segons correspongui (a una ISR s'apaga, a l'altre s'encén).

L'altre cosa a destacar d'aquest exemple és el que hi ha dins el bucle infinit, que està buit. I està buit perquè, en aquest exemple, el microcontrolador no té res a fer fins que no hi hagi una interrupció provinent d'un botó.

En una aplicació real, en aquest bucle es podrà posar codi que si s'hagi d'executar contínuament, o instruccions que posin "a dormir" el microcontrolador tot esperant una interrupció, etc. Tot això ho anirem veient més endavant.

7.1 Escrivint ISRs en C

Com ja sabem, les ISR són les funcions especials que s'executen tant bon punt es dispara una interrupció determinada.

Tradicionalment les adreces a aquestes ISRs (anomenats de vegades vectors d'interrupció) s'emmagatzemaven a una zona especial de la memòria del processador. Quan el processador rebia una IRQ, com que aquestes van numerades simplement calcula l'offset de la IRQ a la taula de ISRs i executa aquella funció determinada.

En els ARM Cortex amb els que treballem això es fa tal qual acabem d'explicar. En el cas dels

```
    .section    .vectors
    .align      2
    .globl      __Vectors
__Vectors:
    .long       __StackTop              /* Top of Stack */
    .long       Reset_Handler           /* Reset Handler */
    .long       NMI_Handler             /* NMI Handler */
    .long       HardFault_Handler       /* Hard Fault Handler */
    .long       MemManage_Handler       /* MPU Fault Handler */
    .long       BusFault_Handler        /* Bus Fault Handler */
    .long       UsageFault_Handler      /* Usage Fault Handler */
    .long       Default_Handler         /* Reserved */
    .long       Default_Handler         /* Reserved */
    .long       Default_Handler         /* Reserved */
    .long       Default_Handler         /* Reserved */
    .long       SVC_Handler             /* SVCall Handler */
    .long       DebugMon_Handler        /* Debug Monitor Handler */
    .long       Default_Handler         /* Reserved */
    .long       PendSV_Handler          /* PendSV Handler */
    .long       SysTick_Handler         /* SysTick Handler */

    /* External interrupts */

    .long       DMA_IRQHandler      /* 0 - DMA */
    .long       GPIO_EVEN_IRQHandler    /* 1 - GPIO_EVEN */
    .long       TIMER0_IRQHandler    /* 2 - TIMER0 */
    .long       USART0_RX_IRQHandler    /* 3 - USART0_RX */
    .long       USART0_TX_IRQHandler    /* 4 - USART0_TX */
    .long       ACMP0_IRQHandler    /* 5 - ACMP0 */
    .long       ADC0_IRQHandler    /* 6 - ADC0 */
    .long       DAC0_IRQHandler    /* 7 - DAC0 */
    .long       I2C0_IRQHandler    /* 8 - I2C0 */
    .long       GPIO_ODD_IRQHandler    /* 9 - GPIO_ODD */
    .long       TIMER1_IRQHandler    /* 10 - TIMER1 */
    .long       USART1_RX_IRQHandler    /* 11 - USART1_RX */
```

Figura 7.1: Vectors d'interrupció

Cortex (i la majoria de microcontroladors i processadors) la taula de vectors d'interrupcions es col·loca a partir de la posició 0 de memòria.

El que cal, doncs, és que les nostres eines de compilació posin aquests vectors com toca a cada un dels binaris que generem. En el cas de les eines per Cortex (tant Simplicity com les eines de ST ho fan així), aprofiten un codi d'inicialització proporcionat per ARM anomenat, en el nostre cas startup_efm32tg.S. Aquest fitxer està escrit en assemblador i, entre d'altres coses, té el codi que es veu a la Figura 7.1:

Com es pot veure, aquest codi declara el nom de les ISRs corresponents a cada una de les IRQs possibles al microcontrolador. Més endavant en el mateix fitxer, es posa una funció per defecte per a cada una de les ISR que és només un bucle sense fi. Com que aquesta funció es declara com *weak*, nosaltres podrem sobreescriure–la en cas que ho vulguem fer.

En el cas de Cortex, les funcions de ISR no han de ser definides de cap forma especial, més enllà de posar el nom que li correspongui.

En altres arquitectures, com ara AVR d'Atmel, les ISR han de retornar d'una manera diferent a les funcions normals donat que durant la crida a una ISR no es guarden tots els registres com es fa a una crida a una funció normal. Si escrivim la funció en assemblador, enlloc d'una instrucció **RET** ens caldrà escriure una instrucció **RETI**.

Si estem treballant en llenguatge C, com que el compilador posa una instrucció **RET** quan acaba la funció, caldrà indicar-li d'alguna forma que la funció en qüestió és una ISR i que ha d'acabar-la

Llistat 7.2: Exemple d'ISR per AVR

```
#pragma vector=TIMER0_OVF_vect
__interrupt void MotorPWMBottom() {
  // codi
}
```

Llistat 7.3: Exemple d'ISR per AVR

```
ISR(PCINT1_vect) {
  //codi
}
```

amb una instrucció **RETI**.

Això, en el cas d'AVR es fa com es veu al Llistat 7.2 o 7.3 depenent del tipus de compilador que estiguem fent servir.

7.2 Fent servir ISRs

Com ja hem comentat, una ISR s'executa quan es dispara una IRQ. Durant l'execució de la ISR el microcontrolador està en un mode d'execució especial, amb les demés IRQs desactivades (depèn de l'arquitectura això pot no ser així).

Donat que la resta d'IRQs poden estar desactivades, és important que el temps que el processador estigui executant una ISR sigui el mínim possible i que el codi, per tant, sigui el més senzill possible.

Si el que hem de fer, a part de tasques molt simples a la ISR, és engegar o controlar un procés més complicat, aquest procés no el farem dins la ISR, si no en un procés a part i comunicarem via *flags*, cues, semàfors o mecanismes similars la ISR amb el procés. Així minimitzem el temps que el processador està en mode ISR.

7.2.1 Ús de variables globals

Com ja hem vist breument a **Subsecció 2.3.1 - Perifèrics mapats a memòria** hi ha una paraula reservada que es fa servir quan es fan accessos a memòria i altres usos d'una variable on el compilador no ha d'actuar amb cap optimització. La paraula reservada **volatile** davant la declaració d'una variable indica al compilador que la variable s'hi ha d'accedir tal com diu el codi i no efectuar cap optimització.

R En el cas de modificar el valor d'una variable global des d'una ISR, cal declar-la com a **volatile**.

8. Timers

Un *Timer* (temporitzador) és un dels perifèrics més habituals de trobar en un microcontrolador. Bàsicament consisteix en un comptador que pot generar alguna interrupció quan arriba a un cert llindar o al seu valor límit. Com sempre, cada fabricant el fa segons el seu criteri i, per tant, cada un té característiques diferents.

Normalment els *timers* es poden connectar a diferents rellotges disponibles dins el microcontrolador. A més, força sovint els *timers* poden dividir prèviament la freqüència del rellotge que l'alimenta per reduir-la encara més. Així, podem tenir un rellotge d'1 MHz alimentant un *timer* que abans de que hi entri es divideixi per 8 per tenir un rellotge efectiu de 125 kHz. Amb això, si configurem el *timer* perquè compti fins al valor 125000, tindrem que el *timer* generarà una interrupció cada segon.

En el cas de Silicon Labs, els Timers tenen múltiples opcions [24, pàgina 249]:

- comptador de 16 bits
- pre-escalatge del rellotge: el rellotge d'entrada es pot pre-escalar (dividir) per diversos factors (de 2 fins a 1024).
- diverses fonts de rellotge
- diverses formes de comptatge (cap a munt, cap avall, amunt i avall, etc.)
- 3 canals per Timer, per generar diverses interrupcions (per *overflow*, per arribar a un llindar, *underflow*, etc.)

Tot plegat fa que sigui força complicat de configurar, i com ve sent costum, el fabricant ens dona una biblioteca per simplificar-nos una mica la vida.

Els controls que tenim habitualment per un Timer, un cop configurat, són [25]:

- engegar i parar el Timer (**TIMER_Enable()** a EMLIB).
- llegir o configurar el màxim valor pel timer (**TIMER_TopGet() / TIMER_TopSet()** a EMLIB).
- llegir o configurar el valor pel compare (**TIMER_CompareGet() / TIMER_CompareSet()** a EMLIB).

Els usos que li podem donar a aquest perifèric són variats, els més habituals són els següents::

- Comptar el temps: es configura per a que generi una interrupció cada segon i ja tenim un rellotge de temps real. Hi ha perifèrics específics per aquesta tasca (RTCs[1]) com es veurà a **Capítol 9 - RTC**.
- Fer *delays* acurats: de vegades cal que un senyal o una acció passi després d'un cert temps. Amb un Timer ben configurat podem comptar temps petits de l'ordre de microsegons.
- Comptar polsos externes: segons quins Timers poden comptar segons una entrada externa, i aquesta no cal que sigui un rellotge. Pot ser, per exemple, les pulsacions d'un botó o les transicions d'un senyal.
- Generar senyals PWM: tipus de senyal digital per controlar motors o d'altres dispositius (Veure **Capítol 10 - PWM**).

8.1 Exemple senzill amb un Timer

El primer exemple fa servir un Timer per esperar-se 1 segon a canviar d'estat el LED (fer un *toggle*) després que es premi el Botó 0.

El que es fa en primer lloc és configurar el Timer0 amb un seguit d'opcions. Les més importants de cara a l'exemple són:

- .prescale = timerPrescale1024 que configura el divisor de rellotge a 1024.
- .mode = timerModeUp així el Timer només compta cap amunt.
- .oneShot = true en aquest cas, un cop arribi al màxim valor el Timer es pararà.

Un cop configurat el Timer, s'entra en el bucle infinit. Dins el bucle infinit, si es polsa el botó 0, es posa el comptador del Timer a 0 i s'engega el comptador (Llistat 8.1).

Tot seguit, es comprova si el comptador ha arribat al valor **TOP_VALUE** usant la funció **TIMER_CounterGet()**, si és així es fa *toggle* del LED (s'encén si estava apagat o el contrari), s'atura el Timer i es posa el seu comptador a 0 (Llistat 8.2).

Com hem calculat el valor **TOP_VALUE** (13671)?

```
Rellotge d'entrada al Timer: 14.000.000 Hz (14 MHz)
Prescaler: 1024 (Seleccionat al inicialitzar el timer)
Freqüència de treball del Timer = 14.000.000 / 1024 = 13.671,875 Hz
```

Per tant, en un segon el comptador del Timer haurà comptat fins a 13.671 (o 13.672, no ve d'un tick!).

[1] Real Time Clocks

Llistat 8.1: Codi d'exemple d'ús d'un Timer

```c
void main(void) {
  ...
  if (GPIO_PinInGet(gpioPortD, 8) == 0) {
    TIMER_CounterSet(TIMER0, 0);
    TIMER_Enable(TIMER0, true);
  }
}
```

Llistat 8.2: Codi per comprovar si el Timer ha arribat a cert valor

```
void main(void) {
  ...
  /* If timer count gets to TOP_VALUE, toggle LED and stop Timer */
  if (TIMER_CounterGet(TIMER0) >= TOP_VALUE) {
    GPIO_PinOutToggle(gpioPortD, 7);
    TIMER_Enable(TIMER0, false);
    TIMER_CounterSet(TIMER0, 0);
  }
  ...
}
```

En aquest exemple hem fet servir el Timer d'una forma força rudimentària, ja que no és gaire habitual fer *polling* d'un Timer per transcorre un temps determinat. Es fa servir aquest mètode per implementar funcions tipus **Delay()** simples. A continuació veurem un exemple més complicat basat en interrupcions.

8.2 Exemple més complex amb el Timer

Al segon exemple fem servir interrupcions per obtenir informació del Timer i així alliberar la CPU.

Primer de tot, cal saber per quines condicions pot generar interrupcions el nostre Timer. En el cas de la família EFM32, cada Timer té només una interrupció (anomenada TIMERn_IRQn) però tenen els següents esdeveniments que poden generar una interrupció [24]:

- *Overflow*: quan el comptador arriba a **TOP**
- *Underflow*: quan el comptador arriba a 0
- *Compare Match*: quan el comptador arriba a un valor determinat. N'hi ha un per cada canal del Timer.

Així doncs, quan estiguem a la ISR del Timer si hem activat més d'un esdeveniments a que generi la interrupció haurem de mirar quin esdeveniments ha estat.

El Timer es configura igual que a l'exemple anterior, i a més s'activen les interrupcions per aquest perifèric amb el codi que es veu a 8.3.

Amb això, un cop s'engegui el Timer i quan arribi el comptador a **TOP** llençarà la interrupció **TIMER0_IRQn**, que executarà la **ISR TIMER0_IRQHandler()** que es veu al Llistat 8.4 on simplement es netegen els *flags* d'interrupció i es fa *toggle* del LED.

La configuració i la posada en marxa del Timer es fa a la ISR del botó 0, de manera similar a com ja havíem fet anteriorment a d'altres exemples: primer es netegen els *flags* de la IRQ, tot seguit es

Llistat 8.3: Codi corresponent a l'activació de les IRQs del Timer

```
 /* Enable overflow interrupt */
TIMER_IntEnable(TIMER0, TIMER_IF_OF);

/* Enable IRQ for Timer 0*/
NVIC_EnableIRQ(TIMER0_IRQn);
```

Llistat 8.4: ISR del Timer

```
void TIMER0_IRQHandler(void) {
  uint32_t flags;

  /* Clear flag for TIMER0 */
  flags = TIMER_IntGet(TIMER0);
  TIMER_IntClear(TIMER0, flags);

  /* Toggle LED ON/OFF */
  GPIO_PinOutToggle(gpioPortD, 7);
}
```

Llistat 8.5: ISR del GPIO per l'exemple del Timer

```
void GPIO_EVEN_IRQHandler(void) {
  uint32_t flags;

  /* clear flags */
  flags = GPIO_IntGet();
  GPIO_IntClear(flags);

  /* Set counter to 0  */
  TIMER_CounterSet(TIMER0, 0);

  /* Set TIMER Top value */
  TIMER_TopSet(TIMER0, TOP_VALUE);

  /* Start Timer */
  TIMER_Enable(TIMER0, true);
}
```

posa el comptador del Timer a 0, es posa el valor màxim i per últim s'engega el Timer (Llistat 8.5).

Per últim, fer notar que al bucle infinit final del **main()** no hi ha cap codi, ja que la CPU no té res a fer mentre espera la pulsació del botó o que s'exhaureixi el temps, En el tema de baix consum (**Capítol 35 - Baix cosum**) es veurà com aprofitar aquest fet per reduir el consum del sistema amb un exemple a **Secció 35.4 - *Timers* de baix consum**.

Al diagrama de seqüència de la Figura 8.1 explica l'exemple.

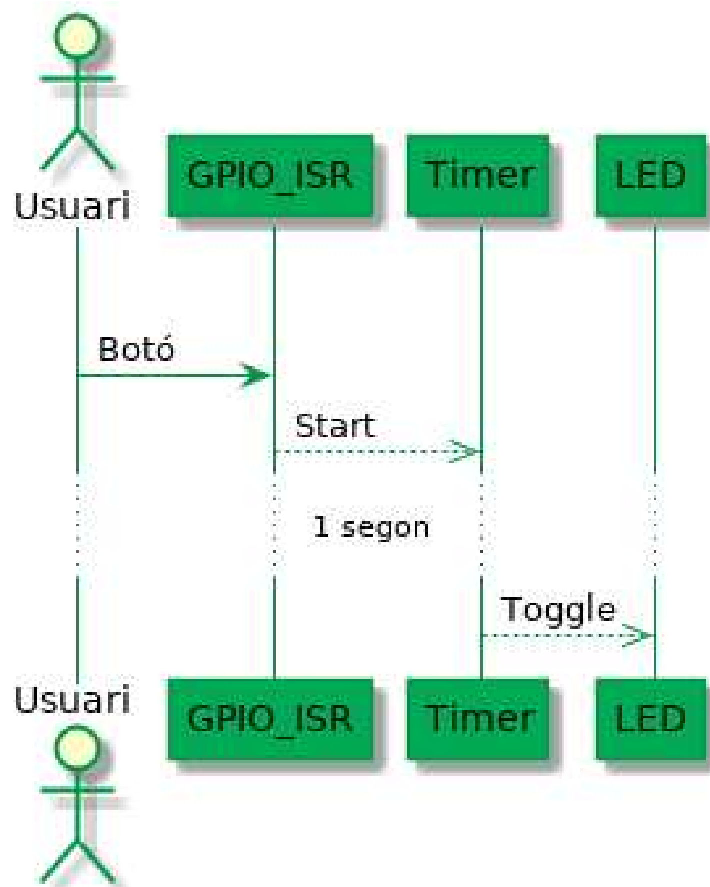


Figura 8.1: Diagrama de seqüència de l'exemple Timer_2

Aquesta pàgina està en blanc expressament, tot va bé.

9. RTC

Un altre perifèric que acostumem a trobar als microcontroladors actuals és una mena de Timer una mica especial. Habitualment aquests perifèrics serveixen per tenir un control de temps en segons i/o un calendari, enlloc de temps molts més curts de mil·lisegons o microsegons com els Timers que ja hem vist.

Aquest tipus de perifèrics acostumen a fer servir una entrada de rellotge pròpia de 32,768 kHz (32.768 Hz), que és una freqüència de rellotge molt habitual per aquestes feines. Algunes famílies de microcontroladors poden funcionar amb altres freqüències o generar-la internament per fer el sistema més senzill.

Els RTC varien força de fabricant a fabricant, així els STM32 tenen un RTC complet, on podem guardar dia, mes i any, hora minut i segons i el dispositiu mateix manté la data (dies del mes, anys de traspàs, etc.), fent molt senzill mantenir una data dins el dispositiu (veure Figura 9.1) [30, pàgina 799].

Per contra, els RTCs de EFM32 són força més senzills, i és, de fet, un Timer de 24 bits de molt baix

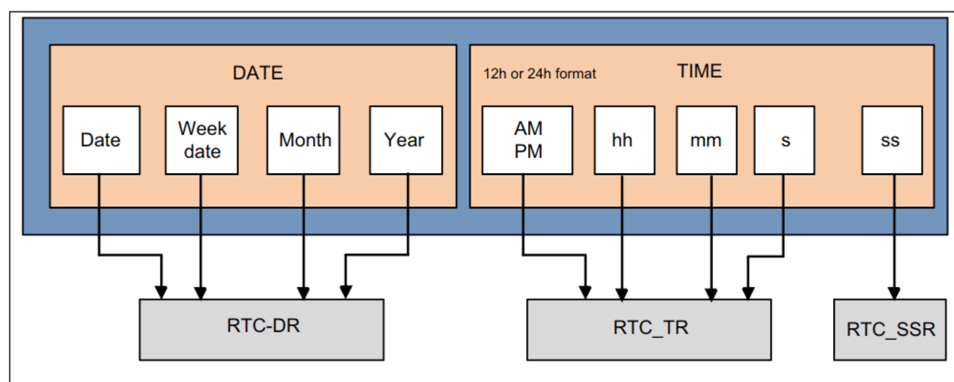


Figura 9.1: Registres del RTC de STM32 [32]

Llistat 9.1: Inicialització del RTC

```
void main(void) {
  ...
  CMU_ClockSelectSet( cmuClock_LFA, cmuSelect_LFXO );
  CMU_ClockDivSet( cmuClock_RTC, cmuClkDiv_32768 );
  ...

  RTC_CompareSet(0, 2);

  /* Enabling Interrupt from RTC */
  RTC_IntEnable(RTC_IFC_COMP0);
  NVIC_ClearPendingIRQ(RTC_IRQn);
  NVIC_EnableIRQ(RTC_IRQn);
  ...
}
```

Llistat 9.2: ISR del RTC

```
void RTC_IRQHandler(void) {
  /* Clear interrupt source */
  RTC_IntClear(RTC_IFC_COMP0);

  GPIO_PinOutToggle(gpioPortD, 7);
}
```

consum amb una entrada de rellotge pròpia i que es pot triar cada quan generen una interrupció [24, pàgina 285]. Si triem fer una interrupció cada segon, podem manegar per SW la gestió de l'hora i el calendari.

A l'exemple per EFM32 tant sols se selecciona el rellotge de 32 kHz extern (LFXO), el divisor a 32768 per tenir un tic cada segon i es configura per a que generi una interrupció cada 2 segons (veure Llistat 9.1). En aquest exemple, a la ISR només es canvia el LED (s'apaga o encén segons el seu estat actual) (veure Llistat 9.2).

Si volguéssim mantenir un rellotge fent servir aquest perifèric podríem configurar-lo perquè generi una interrupció cada segon, i dins la ISR mantenir un comptador de segons i actualitzar la data segons això.

9.1 RTC externs

Quan la majoria de microcontroladors no incloïen un RTC intern com els que hem vist, era habitual fer servir un dispositiu RTC extern. D'aquests dispositius n'hi ha de tota mena però la majoria tenen les següents característiques:

- Interface I2C[1] (veure **Capítol 15 - I2C**) amb el microcontrolador.
- Necessita un cristall de 32.768 Hz.
- Un error d'aproximadament un segon a l'any.
- Actualitza data i hora, calculant anys de traspàs.
- Capacitat de generar interrupcions segons una alarma programable.

[1] Inter-Integrated Circuit

- Molt baix consum i alimentació separada amb bateria (pila botó).
- Molts d'ells tenen una petita memòria RAM adreçable per guardar-hi dades persistents.

Amb aquesta mena de dispositiu, el microcontrolador es descarrega de gestionar el calendari i només cal accedir als registres del RTC extern per saber l'hora o data del sistema. A la majoria dels casos també és possible programar alguna mena d'alarma, de forma que quan arriba cert temps o data una línia dedicada pot generar una interrupció al microcontrolador. Alguns models també tenen la possibilitat de generar un senyal de forma periòdica per tenir, per exemple, un senyal a 128 Hz.

Habitualment l'alimentació d'aquests dispositius es pot fer per un canal separat de l'alimentació principal i usant una bateria o pila tipus botó. Això permet que el RTC sempre estigui alimentat encara que es perdi l'alimentació principal (per avaria, tall de corrent, canvi de bateries, etc.). Aprofitant que sempre tenen alimentació, força RTCs tenen una zona de memòria RAM per a que el microcontrolador pugui guardar-hi dades persistents. L'accés a aquesta zona de memòria també es fa mitjançant el bus I2C, sent molt senzill emmagatzemar-hi dades.

Per últim, cal dir que són dispositius força barats (entre 0.5 € i 3 € per unitat en volums petits), sent una bona opció en cas que el nostre microcontrolador no disposi d'aquesta funcionalitat [33][34][35].

Aquesta pàgina està en blanc expressament, tot va bé.

10. PWM

El PWM és una tècnica per aconseguir controlar la potència subministrada a un dispositiu mitjançant un senyal digital. Simplificant, fent que un senyal digital ('1' o '0') estigui més o menys estona a '1' aconseguim controlar la potència que rep el dispositiu a la sortida.

Aquest tipus de modulació es fa servir per controlar motors simples (motors DC) on enlloc d'enviar un voltatge variable per controlar la velocitat enviem un senyal PWM. Així, si enviem polsos més llargs el motor girarà més ràpid i si enviem polsos més curts el motor girarà més a poc a poc. Com que el voltatge que se li envia sempre es el màxim, la potència del motor és sempre la màxima.

Els dos paràmetres principals d'un senyal PWM són la seva freqüència i el seu duty cycle (la durada del pols a '1' respecte la durada del pols a '0').

A l'exemple es fa servir el LED de la placa enlloc d'un motor. Posarem una freqüència molt petita, així la podrem veure amb els nostres ulls, i anirem canviant el duty cycle amb els dos botons, de manera que podem veure què està passant.

Primer veiem què passa i després veiem com ho fa el codi.

Només programar la placa veiem el LED fent pampallugues força ràpid tota l'estona. Si polsem el botó 1 veurem que el LED va més ràpid, si tornem a polsar el botó 1 encara va més ràpid així fins que al cinquè cop que el polsem el LED es queda encès tota l'estona.

El que estem veient és que el LED va rebent un senyal PWM on el duty cycle cada cop és més gran (més estona encès que apagat) fins que al final és del 100% (sempre encès). El ritme al que fa pampallugues el LED és (més o menys) la freqüència del PWM, que en aquest exemple és de 13.5 Hz.

10.1 Generar PWM

La majoria de microcontroladors actuals tenen algun dispositiu HW que permet generar PWM. En el cas dels micros de Silicon Labs, el dispositiu que ens permet generar-ne són els Timers.

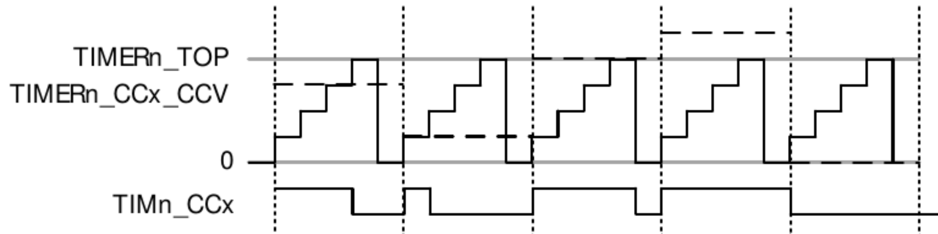


Figura 10.1: Generació de PWM amb un timer [24, pàgina 262]

Llistat 10.1: Configuració del Timer per l'exemple PWM

```
/* Set Timer */
TIMER_InitCC(TIMER1, 1, &timerCCInit);
TIMER1->ROUTE |= (TIMER_ROUTE_CC1PEN | TIMER_ROUTE_LOCATION_LOC4);
```

Un Timer (veure la secció **Capítol 8 - Timers**) no és més que un comptador HW que genera interrupcions o sortides quan el comptador arriba a uns certs valors.

Per generar PWM, el Timer es configura el seu registre **TOP** per a que ens doni la freqüència de funcionament del PWM desitjada. El que farà el Timer és comptar sempre fins a TOP i re-iniciar-se en quan hi arribi. Per generar el duty cycle el que es fa es posar el valor desitjat al registre **COMPARE** (**CC**). El que farà el Timer és treure un '1' mentre el comptador no arribi a **CC**, llavors posarà un '0' a la sortida fins que el comptador arribi a **TOP**, on tornarà a començar el cicle (Figura 10.1)..

En el codi d'exemple (Llistat 10.1), el que es fa és configurar el TIMER1 en mode PWM i connectar la sortida del Timer al Pin D7 que és on està el LED connectat.

A continuació és configuren els dos registres importants per generar PWM (Llistat 10.2). Com que volem una freqüència baixa pel PWM per poder veure'l amb els nostres ulls, al registre **TOP** hi posem **PWM_FREQ** (4000), que el calculem de la següent manera [24].

$$\text{Freq. de PWM} = \frac{\text{Freq Clk}}{\text{prescaler} * \text{valor TOP}} = \frac{14.000.000\text{ Hz}}{256 * (4000+1)} = 13.67\text{ Hz}$$

La resta del codi és senzill: es preparen les dues interrupcions per cada un dels botons, i quan algun dels dos es prem, la ISR modifica el valor del registre **CC** del Timer (un botó augmenta el duty cycle, l'altre el disminueix).

Si veiem el senyal generat amb un oscil·loscopi veiem el que es mostra a la Figura 10.2 (estat per defecte).

Veiem que la freqüència real del senyal és de 13.57 Hz (73.70 ms de període) i que el senyal està a '1' 12.30 ms i a '0' a 61.40 ms.

Llistat 10.2: Configuració del Timer per l'exemple PWM

```
TIMER_TopSet(TIMER1, PWM_FREQ);
TIMER_CompareBufSet(TIMER1, 1, pwm_value);
```

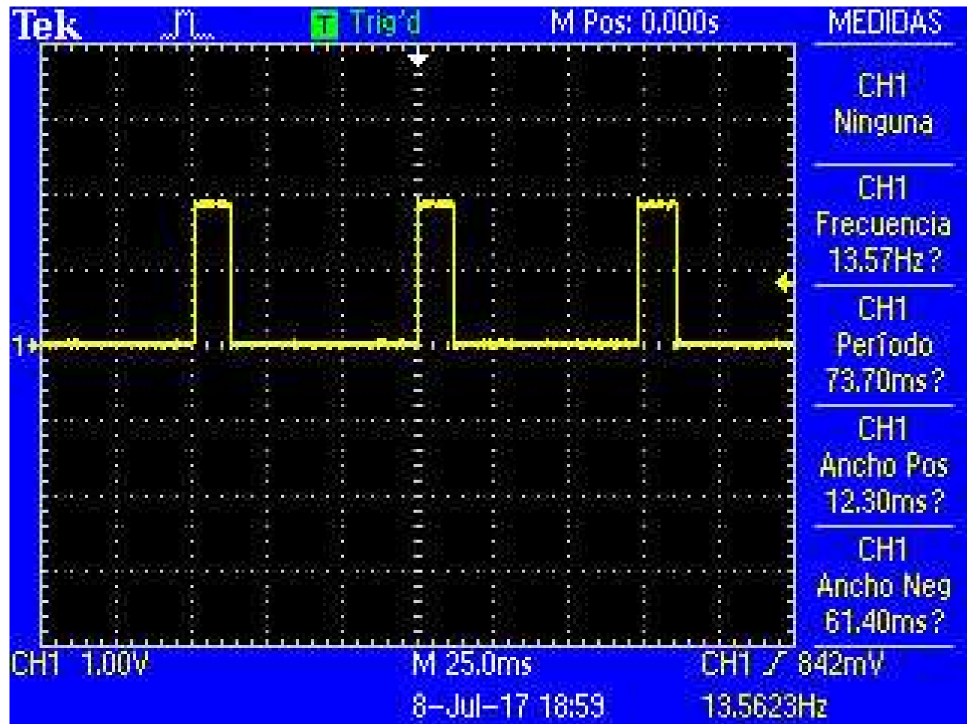


Figura 10.2: PWM amb Duty Cycle al 16%

Segons anem pitjant el botó 1 i anem augmentant el duty cycle anem veient com està més estona a '1' el senyal (Figures 10.3, 10.4 i 10.5). Cal fixar-se que la freqüència no varia, sempre és ˜13.5 Hz i el que va variant és l'estona que està a '0' o a '1' el senyal.

Per acabar, es pot provar de canviar la freqüència del PWM per a que no es vegi el LED fent pampallugues. Cal augmentar la freqüència a un valor que superi els 100Hz i enlloc de veure el LED encendre's i apagar-se, es veurà com varia la intensitat amb la que llueix.

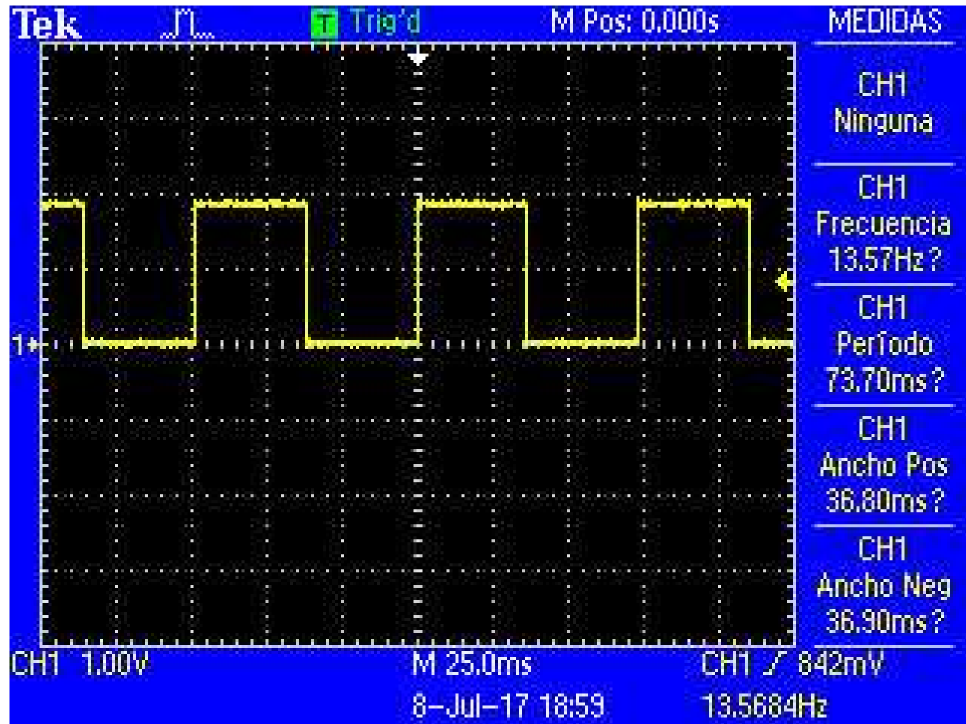


Figura 10.3: PWM amb Duty Cycle al 50%

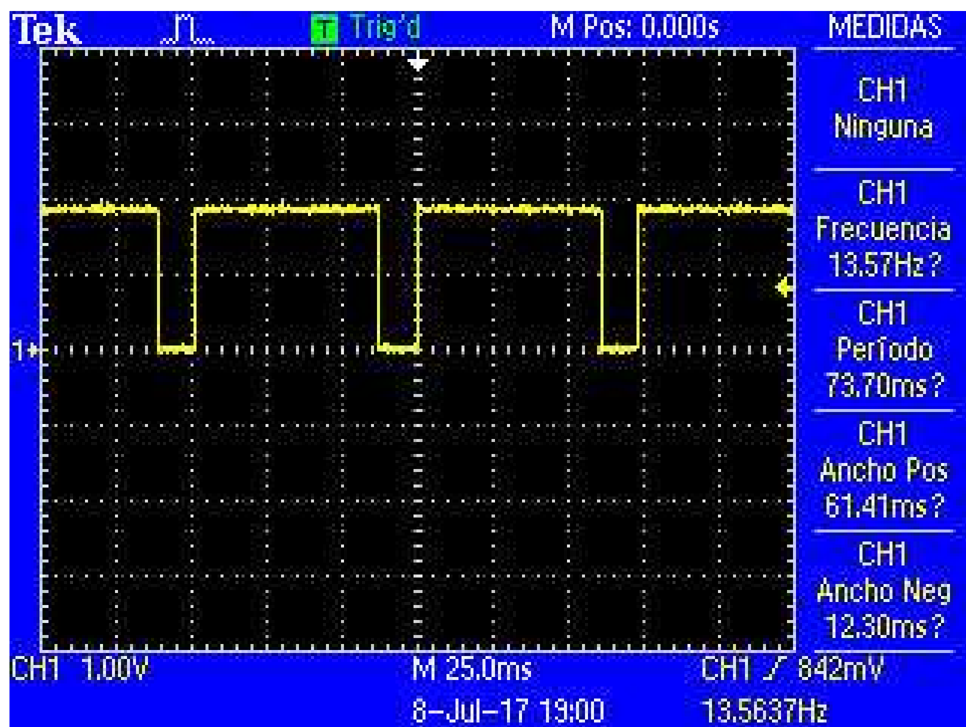


Figura 10.4: PWM amb Duty Cycle al 83.3%

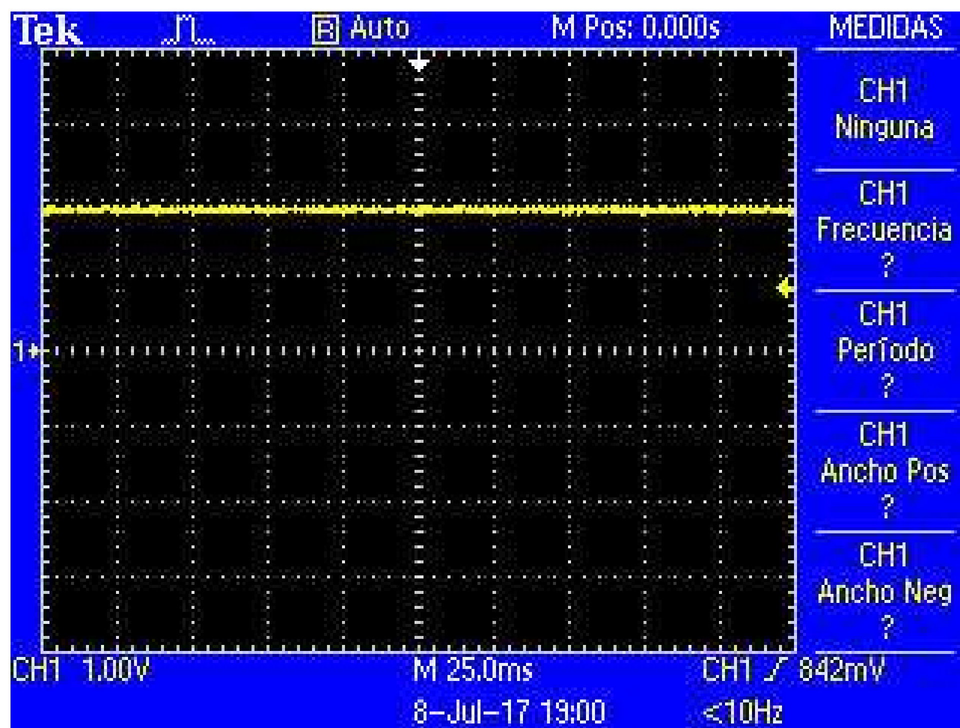


Figura 10.5: PWM amb Duty Cycle al 100%

10.2 Controlant un servomotor

Un servomotor és un dispositiu electromecànic de control senzill. La majoria d'ells son rotatius i permeten controlar l'angle d'actuació del motor amb un senyal digital, normalment un senyal PWM. Així, el que se sol necessitar és un senyal PWM a una freqüència determinada i un temps actiu entre certs valors que provocaran un moviment proporcional al motor. En el cas del servo que tinc entre mans, un Parallax Standard Servo (#900-00005) (veure Figura 10.6) (enllaç al seu DataSheet), cal un PWM a 50 Hz i uns temps mínims de 0.75 ms i màxim de 2.25 ms. Aquests temps faran que el servo es mogui entre els 0º i els 180º de rotació.

Cal aplicar la següent fórmula per saber el valor TOP del nostre TIMER:

$$TOP = \frac{f_{CPU}}{f_{PWM} * PRESCALER} - 1$$

Com que la freqüència d'entrada és 14.000.000 Hz, la freqüència del PWM ha de ser 50 Hz i TOP no pot ser més gran de 65.536 (16 bits) cal que triem el *prescaler* acuradament. De fet, podríem triar un prescaler de 1024 i ens donaria un valor per TOP de 272. Però amb aquest valor perdem molta resolució per generar el PWM, ja que el *timer* només podrà arribar fins a aquest valor. Si triem un valor més petit pel prescaler, augmentem la resolució i la qualitat del senyal del PWM. Així, si triem un valor de pre-escalat de 8 ens resulta que TOP ha de valor 34999. Aquest valor és el més gran que podem generar d'aquesta forma i que càpiga en 16 bits, ja que el següent pre-escalat ens dona un valor massa gran.

Així doncs, tenim que per generar un PWM 50 HZ calen 34999 *ticks* del *timer*. Com que la fulla d'especificacions del servomotor ens diu que el temps de pols positiu ha d'anar dels 0.75 ms fins els 2.25 ms, cal saber quants *ticks* els correspont.

Figura 10.6: Fotografia del servomotor Parallax usat

Si fem una simple regla de tres tenim:

$$0^\circ \rightarrow \frac{0.75\ ms}{20\ ms} * 34.999 = 1312,4625 \simeq 1312$$

i

$$180^\circ \rightarrow \frac{2.25\ ms}{20\ ms} * 34.999 = 3937,3875 \simeq 3937$$

I ja tenim els dos valors màxims i mínims per controlar correctament el servomotor en qüestió i ens podem muntar una funció que ens passi de graus de rotació a *counts* del PWM per simplificar el codi (Llistat 10.3).

A les figures 10.7 i 10.8 es veu el senyal PWM generat pels casos de 0°i 180°. Es veu que les mesures de l'oscil·loscopi donen 50 Hz i un temps del pols positiu de 0.5 ms i 2.25 ms.

I posem un codi a les ISR dels dos botons perquè incrementi (decrementi) el valor en graus on volem situar el servo: La resta del codi està al repositori en aquest enllaç.

R En el meu cas i amb el servomotor que tinc, s'observa que quan està en repòs de tant en tant se sent com el servo fa algun moviment o "crec". No n'estic segur, però pot ser que sigui perquè la freqüència del PWM no sigui prou estable i s'escurci un pel i se surti d'especificacions. En aquest cas millora si s'allarga el període del PWM augmentant una mica el valor de PWM_FREQ.

R També veig que si el mínim el poso a 0.75 ms el servo no arriba a fer 180º de rotació i puc baixar el mínim fins a 0.5ms i que tot segueixi funcionat. En aquest cas el valor de PWM_0 és 875.

Llistat 10.3: Funció que calcula els *counts* donat els graus que es vol del servomotor

```
#define PWM_FREQ (34999)
#define PWM_0 (1312)
#define PWM_180 (3937)

uint32_t degrees_to_pwm(uint32_t degrees) {
    uint32_t ret_value = 0;

    if (degrees < 180) {
        ret_value = (degree * (PWM_180 - PWM_0) / 180 ) + PWM_0;
    } else {
        ret_value = PWM_180;
    }

    return ret_value;
}
```

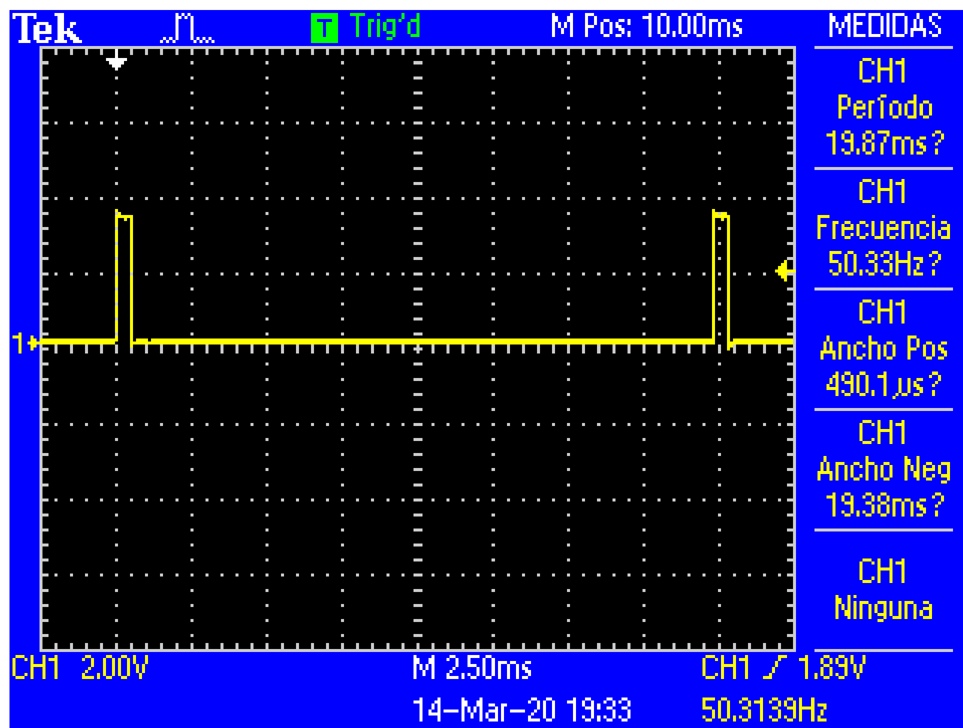


Figura 10.7: PWM per situar el servomotor a 0 °

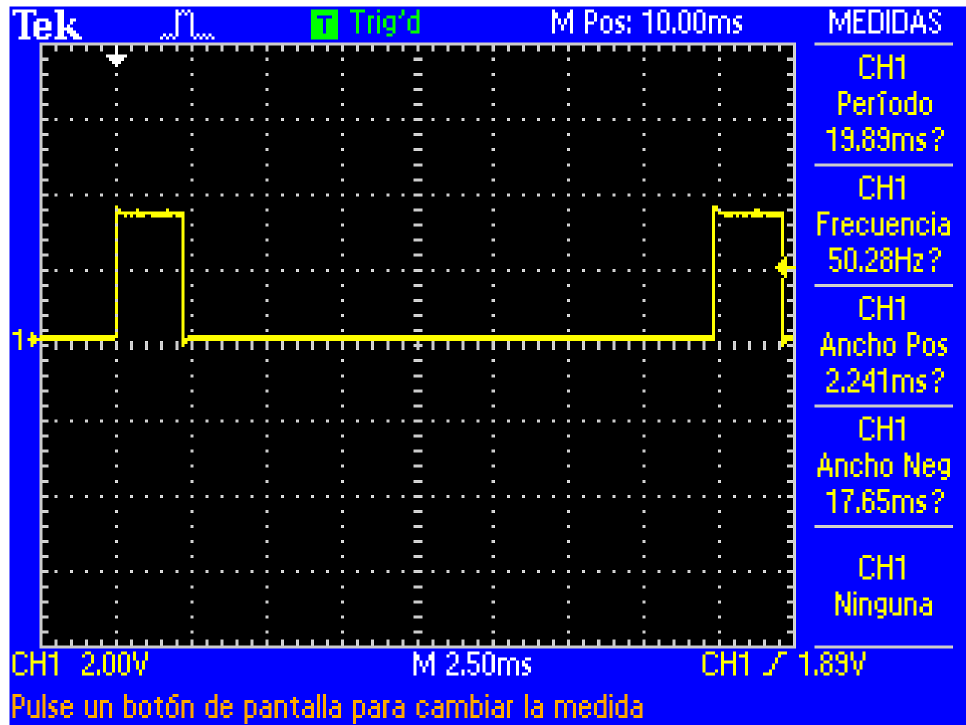


Figura 10.8: PWM per situar el servomotor a 180 °

Llistat 10.4: ISR del botó que incrementa la rotació del servomotor

```
void GPIO_EVEN_IRQHandler(void) {
    uint32_t aux;

    aux = GPIO_IntGet();

    /* clear flags */
    GPIO_IntClear(aux);

    degree += 30;

    if (degree > 180) {
        degree = 180;
    }

    pwm_value = degrees_to_pwm(degree);

    TIMER_CompareBufSet(TIMER1, 1, pwm_value);
}
```

11. *Watchdog*

En els microcontroladors actuals tenim un perifèric amb un funcionament força peculiar. Quan s'activa el watchdog, aquest comença a comptar un cert període de temps, i si no "s'alimenta", reiniciarà tot el sistema [24, pàgina 123][30, pàgina 709].

I per què volem un perifèric que ens reinici el nostre sistema? Doncs per si el nostre FW té algun error i es queda penjat (està en un *Dead-lock*, en un bucle sense sortida, etc.), sempre serà millor que el sistema s'iniciï de nou. Imaginem el cas d'un marcapassos (un sistema encastat força crític); què és millor? Que es quedi penjat per un error del FW que passa molt poc (si passés sovint s'hauria detectat) o que quan passi aquest error el sistema es reinici i torni a funcionar en menys de, posem, un segon?

I com evitem que si tot va be el *watchdog* no ens reiniciï el sistema? Doncs "alimentant-lo" de tant en tant de manera que el comptador intern del *Watchdog* torni a zero (Figura 11.1).

La majoria de fabricants ens donen unes poques funcions per treballar amb el *Watchdog* (en el cas de Silicon Labs a la biblioteca EMLIB tenim el mòdul em_wdog). Habitualment hi ha alguna funció per configurar-lo, una per engegar-lo i una per alimentar-lo. Hi ha fabricants que no permeten deshabilitar el *Watchdog* un cop s'ha engegat per assegurar-se que cap error de FW provocarà que deixi de funcionar.

Habitualment es pot triar quina és la freqüència de funcionament del *Watchdog*, per tenir més o menys temps abans no reiniciï el sistema; normalment de pocs mil·lisegons fins a algunes desenes de segons.

```
WDOG_Feed();
```

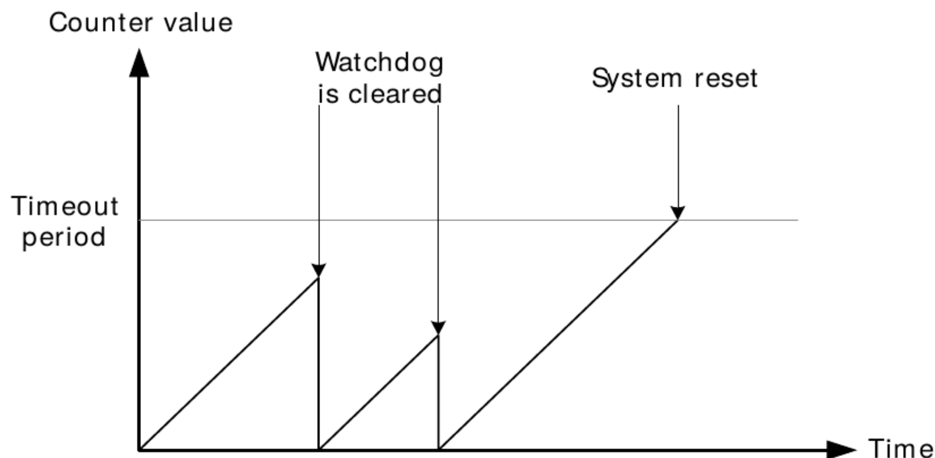


Figura 11.1: Funcionament del em Watchdog [36]

11.1 Exemple

En l'exemple que hi ha al repositori es configura el *Watchdog* perquè treballi amb un rellotge intern de 1 kHz i que compti fins a 4097, de manera que si ningú alimenta el *Watchdog* en 4 segons, aquest reiniciarà el sistema. L'exemple conté un bucle que va fent blinkar el LED de la placa i una ISR que quan es prem el botó 0 alimenta el *Watchdog* (Llistat 11.1).

Si no premem el botó abans no passin 4 segons, el sistema es reiniciarà. I com ho veurem a l'exemple? Doncs perquè el codi el primer que fa és esbrinar per què s'està reiniciant el sistema (Llistat 11.2). Si ha estat perquè ha entrat el *Watchdog*, el LED no blinkarà.

Cal tenir en compte que aquest dispositiu serveix per solucionar possibles fallades totals del sistema, així que cal ser curosos amb el seu ús. Així, si tenim un bucle **for** que pot generar algun problema, no te sentit posar la comanda de *touch* al **watchdog** dins del **for**, si no que segurament te sentit fer-ho abans i després del bucle.

Llistat 11.1: ISR del botó que alimenta el *Watchdog*

```
void GPIO_EVEN_IRQHandler(void) {
  uint32_t aux;

  aux = GPIO_IntGet();
  /* clear flags */
  GPIO_IntClear(aux);

  /* Feed watchdog */
  WDOG_Feed();
}
```

Llistat 11.2: Codi per detectar la causa del reinici

```
if (resetCause & RMU_RSTCAUSE_WDOGRST) {
  resetbyWatchdog = true;
} else {
  resetbyWatchdog = false;
}
```

Aquesta pàgina està en blanc expressament, tot va bé.

III Programació de perifèrics II

12 ADC 75
12.1 Exemple d'ADC

13 DAC 79
13.1 Exemple senzill amb el DAC
13.2 Exemple més complicat amb el DAC

14 UART 85
14.1 Fent servir una USART
14.2 Exemple d'ús d'una UART
14.3 Un exemple amb la UART més complicat

15 I2C 89
15.1 Exemple d'I2C

16 SPI 93

17 DMA 95
17.1 Exemple
17.2 Un exemple amb DMA més complicat

18 FLASH 101
18.1 Un exemple senzill
18.2 *Bootloaders*

19 Mòduls criptogràfics 105
19.1 Xifrant dades amb AES-128

20 Altres perifèrics 109
20.1 *Peripheral Reflex System*

21 Una aplicació completa 117
21.1 Biblioteques
21.2 Funció principal
21.3 Afegint-hi interrupcions

Aquesta pàgina està en blanc expressament, tot va bé.

12. ADC

Un perifèric força habitual en els microcontroladors actuals és l'ADC.

Aquesta perifèric el que fa és llegir un senyal analògic (un voltatge) i convertir-lo a un valor digital (un número). Hi ha diversos models d'ADC amb característiques diferents, però bàsicament les característiques principals d'un ADC són:

- Resolució: fa referència a quants bits dona la conversió de l'ADC. Un ADC de 16 bits, en principi, dona més detall del senyal d'entrada que un ADC de 8 bits. Actualment el més habitual és trobar ADCs d'entre 8 i 16 bits de resolució.
- *Sampling rate*: és la cadència amb la que l'ADC agafa una nova mostra del senyal i la converteix a digital. Els microcontroladors actuals incorporen ADCs que poden arribar al milió de mostres per segon.
- Referència: pot ser que es compari el senyal segons una referència determinada i el ADC ens doni el valor respecte a aquesta referència.

En els microcontroladors moderns, és habitual que davant de l'ADC hi hagi un multiplexor analògic, de manera que es pugui convertir diverses senyals connectades a diferents pins amb un mateix perifèric.

A l'hora de fer servir un ADC, caldrà configurar-lo en els paràmetres de funcionament que necessitem per la nostra senyal.

Com la majoria de perifèrics, l'ADC pot generar una o vàries IRQs segons certes condicions, habitualment quan s'ha acabat la conversió. D'aquesta manera la CPU no cal que faci *polling* del registre d'estat per saber si la conversió ha finalitzat.

L'ADC acaba per donar-nos un valor dins el seu rang de treball proporcional al valor de voltatge d'entrada, aquest valor s'acostuma a anomenar *counts*. Per convertir aquest valor en *counts* al valor de voltatge corresponent, cal aplicar la fórmula **Equació 12.1 - ADC**:

$$V_{ADC} = \frac{counts * V_{max}}{2^N - 1} \tag{12.1}$$

On V_{max} és el voltatge màxim de l'entrada i N el nombre de bits (resolució) de la conversió de l'ADC.[1]

12.1 Exemple d'ADC

Per aquest exemple ens cal per primer cop un petit HW addicional. Farem servir un potenciòmetre que ens donarà una tensió entre Vdd i 0 volts segons el seu recorregut. La sortida d'aquest potenciòmetre la connectarem al pin 16 del connector de Debug de la PCB. Els altres dos pins aniran a 19 i 20 del mateix connector (veure esquemàtic a la Figura 12.1 i fotografia del sistema a Figura 12.2).

Connectat així el potenciòmetre, quan estigui a un extrem del recorregut tindrem 0V a l'entrada de l'ADC i quan estigui a l'altre extrem hi tindrem Vdd[2].

A l'exemple trobem un codi molt senzill, on simplement s'inicia l'ADC amb els paràmetres per defecte i només es canvia el canal d'entrada (el 6) i la tensió de referència (en aquest cas Vdd).

D'aquesta manera, els 12 bits de resolució del ADC serviran per comparar la tensió d'entrada amb els 3.3 V amb els que està alimentat el microcontrolador a la placa.

R Els 3.3 Volts és el voltatge de funcionament de la placa d'avaluació. Els ADC normalment poden mesurar tensions fins a la seva tensió d'alimentació i és per això que en aquest cas podem mesurar fins a 3.3 Volts. Si l'alimentació fos menor, el rang de treball de l'ADC també ho seria. En el cas particular de l'ADC dels EFM32, poden tenir voltatges fixes com a referència [24, pàgina 378].

El codi, un cop inicialitzat el ADC, realitza les accions que es veuen al Llistat 12.1.

El que fan aquests 4 línies és:

- Engegar l'ADC i que comenci una conversió
- Esperar a que la conversió finalitzi consultant el registre STATUS de l'ADC.
- Llegir el valor de la conversió feta per l'ADC.
- Imprimir per la consola de *Debug* el valor de la conversió.

Per tant, el que veurem un cop engeguem l'exemple serà com es van imprimint els valors llegits per l'ADC. Si anem movent el potenciòmetre veurem que els valors van canviant des de 0 fins a 4095.

[1] Això és vàlid per configuracions tipus *single-ended*
[2] Voltatge d'alimentació

Llistat 12.1: Codi de lectura de l'ADC

```
while (1) {
  ADC_Start(ADC0, adcStartSingle);
  while (ADC0->STATUS & ADC_STATUS_SINGLEACT);
  ADCvalue = ADC_DataSingleGet(ADC0);
  printf("ADC Value %lu\r\n", ADCvalue);
}
```

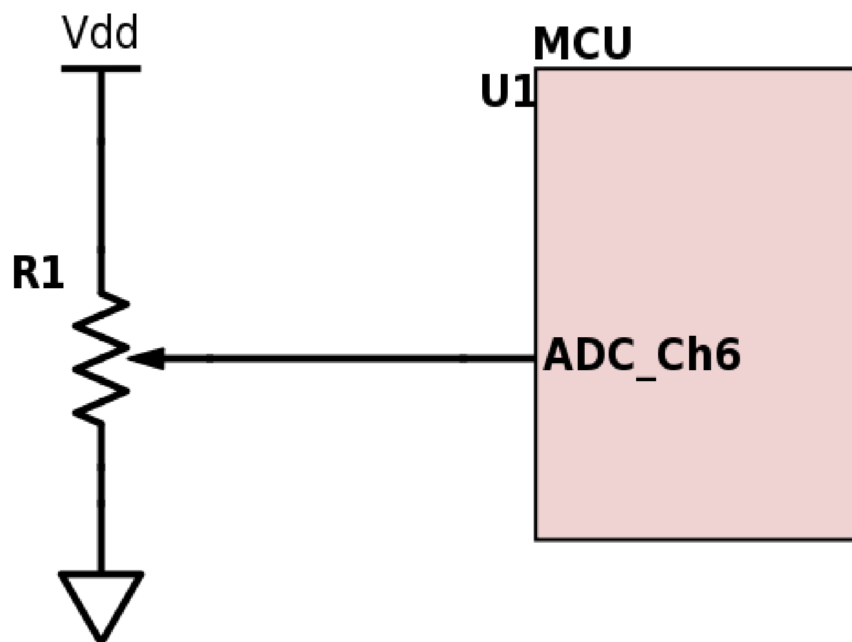


Figura 12.1: Esquemàtic de la connexió del Potenciòmetre al canal d'ADC

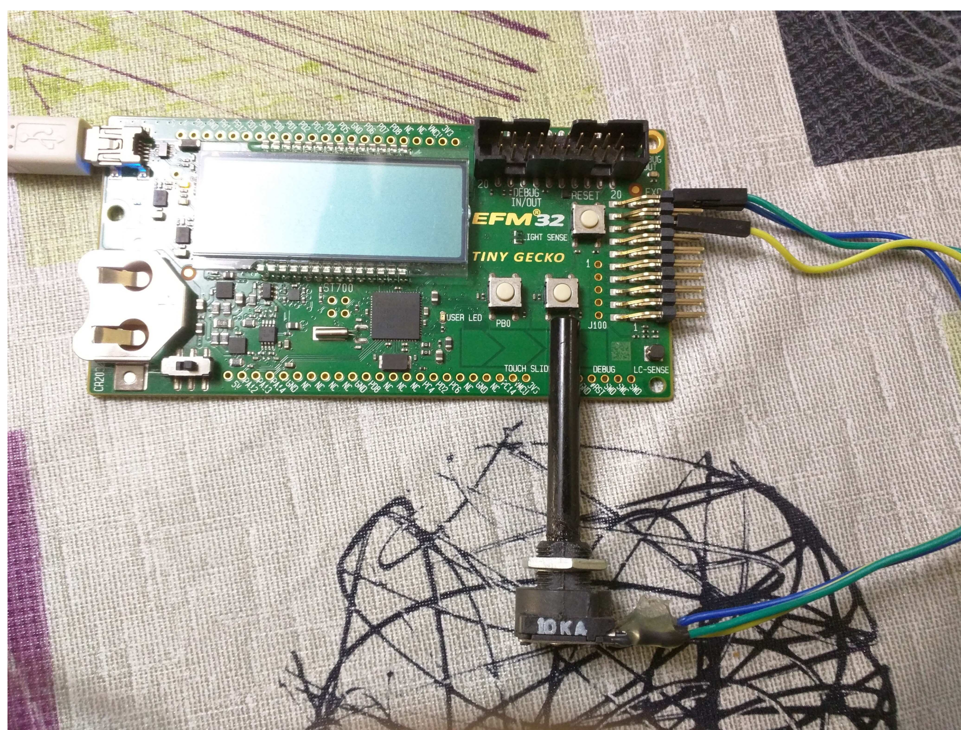


Figura 12.2: Fotografia del sistema amb el connexitat correcte

Aquesta pàgina està en blanc expressament, tot va bé.

13. DAC

Un DAC[1] és un dispositiu que es pot veure com l'invers d'un ADC, ja que a partir d'unes dades digitals genera un senyal analògic equivalent. Els paràmetres de funcionament d'un DAC són, doncs, molt similars als del seu perifèric germà l'ADC.

Al *datasheet* de la família amb la que treballem [24, pàgina 421] hi ha la descripció, bastant breu, de les característiques principals d'aquest perifèric. Bàsicament, té un registre anomenat **CHxDATA** on s'ha d'escriure el valor que volem que es converteixi a voltatge segons la fórmula següent[2]:

$$V_{OUT} = V_{Ref} \times \frac{CHxDATA}{4096} \tag{13.1}$$

On V_{Ref} és el voltatge de referència, que en aquest cas pot ser 2.5 o 1.25 Volts o la tensió d'alimentació Vdd.

13.1 Exemple senzill amb el DAC

Al repositori hi ha un exemple senzill usant el DAC per generar una tensió continua segons un valor donat. Primer s'inicialitzarà el DAC i la resta de perifèrics (en aquest cas els GPIO i poc més). Es configuren els botons com sempre, amb les dues interrupcions ja conegudes (**GPIO_ODD_IRQHandler**() i **GPIO_EVEN_IRQHandler**()).

La part important de l'exemple és la inicialització del DAC, tal com es veu al Llistat 13.1.

En aquest codi, primer es posen els valors per defecte que ens ofereix el fabricant i tant sols es modifiquen alguns paràmetres. Primer s'activa el *clock* del sistema cap al perifèric, tot seguit es tria el *prescaler* segons la freqüència de funcionament desitjada. Després es selecciona que la referència per crear el senyal de sortida es basi en el voltatge d'entrada (**dacRefVDD**). Per últim es configura el DAC i el canal (en aquest el 1 per la sortida que hem triat).

[1]Digital to Analog Converter

[2]Pel cas single-ended, pel cas diferencial veure [24, pàgina 424]

Llistat 13.1: Inicialització del DAC

```
static void DACConfig(void) {
  /* Use default settings */
  DAC_Init_TypeDef init = DAC_INIT_DEFAULT;
  DAC_InitChannel_TypeDef initChannel = DAC_INITCHANNEL_DEFAULT;

  /* Enable the DAC clock */
  CMU_ClockEnable(cmuClock_DAC0, true);

  /* Set prescale for 500 KHz */
  init.prescale = DAC_PrescaleCalc(500000, 0);

  /* Set reference voltage to Vdd */
  init.reference = dacRefVDD;

  /* Initialize the DAC and DAC channel #1 */
  DAC_Init(DAC0, &init);
  DAC_InitChannel(DAC0, &initChannel, 1);
}
```

Llistat 13.2: Bucle infinit del DAC

```
void main(void) {
  while (1) {
    DAC_ChannelOutputSet(DAC0, 1, DACvalue);
    printf("DAC Value %lu (0x%06X)\r\n", (uint32_t) DACvalue, (uint32_t)
        DACvalue);
    while(signal == false);
    signal = false;
  }
}
```

Les ISR dels dos botons el que fan es incrementar o decrementar una variable global que serà el valor que s'enviarà al DAC per a que generi el senyal. Aquest increment o decrement es fa en passos de **DAC_STEP** unitats. D'aquesta manera pitjant els botons podrem seleccionar quin voltatge es genera a la sortida del DAC.

Dins el bucle infinit de la funció **main()** (Llistat 13.2) s'envia el valor de la variable global cap al DAC amb la funció **DAC_ChannelOutputSet()**, es treu el valor per la consola de debug i s'espera a que un *flag* indiqui que hi ha hagut un canvi en el valor de la variable global.

Si mesurem el voltatge de sortida amb un multímetre, oscil·loscopi o analitzador lògic (que tingui entrada analògica) podem veure com els valors de la variable provoquen el canvi en voltatge esperat segons la Fórmula 13.1 tal com es veu a la Taula 13.1. El pin de sortida del DAC està connectat al pin PB12, connectat al pin 13 del connector d'expansió. La Taula 13.1 es registren tots els valors possible de la variable **DACvalue**, el valor teòric que hauria de generar el DAC (segona columna) i el voltatge mesurat amb un multímetre digital (tercera columna); la quarta columna presenta la diferència entre el voltatge de la fila i l'anterior, com es pot veure cada pas corresponent a poc més de 200 mV, tal com surt a la fórmula 13.1. A l'última fila es calcula la mitjana aritmètica de totes aquestes diferències.

L'exemple presentat és força bàsic i serveix per introduir la llibreria **emlib** i el perifèric. A

Taula 13.1: Taula resum dels valors mesurats del DAC

Variable	Voltatge teòric (mV)	Voltatge mesurat (mV)	Pas
0	0	0	-
256	206	204	204
512	413	410	206
768	619	614	204
1024	825	824	210
1280	1031	1032	208
1536	1237	1238	206
1792	1444	1442	204
2048	1650	1651	209
2304	1856	1855	204
2560	2063	2060	205
2816	2269	2260	200
3072	2475	2470	210
3328	2681	2670	200
3584	2888	2880	210
3840	3094	3090	210
4096	3300	3290	200
Mitjana de Pas			205,5

continuació veurem un exemple més complicat.

13.2 Exemple més complicat amb el DAC

En aquest exemple es farà servir el DAC per generar un senyal periòdic tipus triangular (en anglès *Triangle wave*).

Es fa servir un *timer* per tenir una base de temps. El *timer* es configura a 1 kHz, dividint el rellotge d'entrada (14 MHz) entre 16 amb el *pre-escaler* i configurant el valor d'*overflow* a 875 perquè generi un senyal cada 1 mil·lisegon. Així, tindrem un nou valor cada 1 mil·lisegon i el senyal es repeteix cada 33 mostres, per tant tindrem un senyal periòdic de 30,30 Hz aproximadament (veure Equació 13.2).

$$F_{senyal} = \frac{1000}{33} = 30,30Hz \tag{13.2}$$

El DAC es configura de la mateixa manera que a l'exemple anterior, en mode continu, amb referència al voltatge d'alimentació Vdd. El *timer* es configura de manera molt similar a l'exemple vist a **Secció 8.2 - Exemple més complex amb el Timer**, activant les interrupcions per tenir la ISR executant-se cada 1 mil·lisegon.

A la ISR del *timer* es calcula el nou valor pel DAC i s'escriu el nou valor calculat (veure Llistat 13.3).

A l'hora d'executar el codi, si es connecta un oscil·loscopi o analitzador lògic al pin 13 del connector d'expansió de la placa de desenvolupament, que es correspon al pin PB12 del microcontrolador es veurà el senyal com es mostra a la Figura 13.1. Es pot observar com cada esglaó dura 1 mil·lisegon i els increments són de 200 mV aproximadament (Figura 13.2).

En aquest exemple s'ha presentat com generar un senyal periòdic mitjançant el DAC i un *Timer*, aquesta combinació de perifèrics és força habitual per generar senyals d'aquesta mena donada la seva senzillesa i facilitat d'ús.

En cas que els valors del senyal estiguessin pre-calculats i emmagatzemats en una taula, la ISR del *Timer* només caldria agafar el següent valor de la taula.

Llistat 13.3: Part de la ISR del Timer per generar la dada pel DAC

```
void TIMER0_IRQHandler(void) {
  ...
  if (direction_up == true) {
    DACvalue += DAC_STEP;
    if (DACvalue >= 0x1000) {
      DACvalue = 0x0FFF;
      direction_up = false;
    }
  } else {
    DACvalue -= DAC_STEP;
    if (DACvalue < 0) {
      DACvalue = 0;
      direction_up = true;
    }
  }
  DAC_ChannelOutputSet(DAC0, 1, DACvalue);
}
```

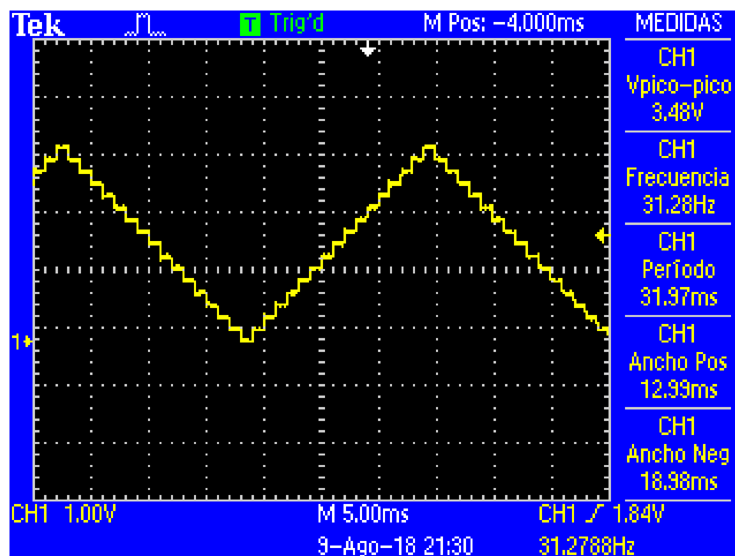


Figura 13.1: Senyal capturat per l'oscil·loscopi.

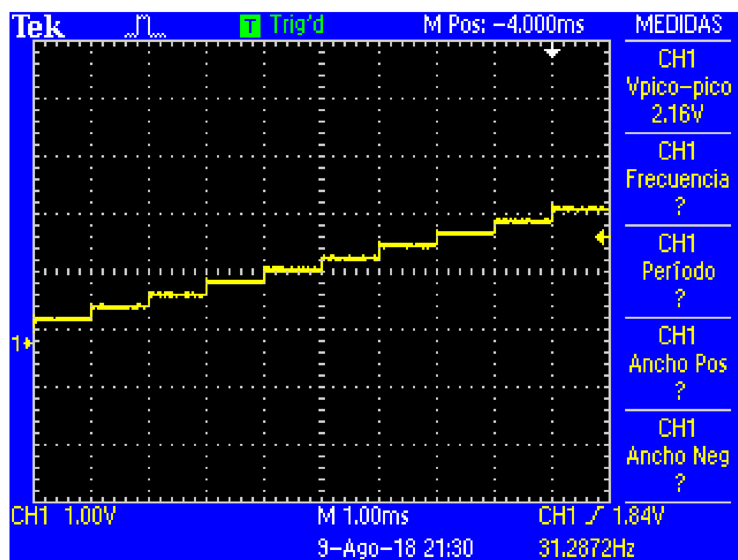


Figura 13.2: Detall del senyal capturat per l'oscil·loscopi
Detall del senyal capturat per l'oscil·loscopi, es pot veure que la unitat de temps és d'1 mil·lisegon.

14. UART

El port sèrie és encara un dels ports d'entrada i sortida més comuns de trobar en un microcontrolador i en un sistema encastat. Encara avui multitud de dispositius fan servir aquest port per rebre o enviar dades i, per tant, els microcontroladors acostumen a incloure uns quants d'aquests ports.

Aquest port sèrie en els microcontroladors el gestiona un perifèric anomenat USART[1] (o UART[2]). El port sèrie habitual, basat en l'estàndard RS-232 només té dos fils, un per rebre dades (RX) i un per enviar-ne (TX). La velocitat i les característiques de la transmissió es poden configurar i habitualment s'envien 8 bits per caràcter amb 1 bit d'stop i sense paritat (tot i que es pot canviar). Les velocitats de transmissió més habituals són: 9600 bps[3], 19200 bps, 56700 bps i 115200 bps. El que cal, com és obvi, és configurar el port sèrie del microcontrolador amb els mateixos paràmetres que el dispositiu extern que estiguem usant.

L'altre aplicació pràctica del port sèrie és poder connectar el microcontrolador a un ordinador personal per poder interactuar amb ell, ja sigui per enviar dades, informació d'estatus o rebre comandes o paràmetres de configuració. Com que actualment la majoria d'ordinadors no tenen un port sèrie, s'han popularitzat molt uns petits dispositius que converteixen el port sèrie en un USB[4] de tipus sèrie. D'aquest dispositiu conegut com CP2102 en parlarem més endavant.

14.1 Fent servir una USART

Com que la configuració d'un port sèrie és una mica complicada, sobretot perquè segons la velocitat de funcionament del rellotge del sistema caldrà dividir i multiplicar aquest senyal fins a tenir una velocitat adequada per la velocitat seleccionada pel port sèrie, els fabricants acostumen a donar-nos biblioteques que ens ajuden.

Dins la biblioteca acostumen a oferir una funció per enviar dades (**USART_Tx()**) i una per rebre'n

[1] *Universal Synchronous and Synchronous Receiver-Transmitter*
[2] *Universal Asynchronous Receiver-Transmitter*
[3] Bits per segon
[4] Universal Serial Bus

Llistat 14.1: ISRs de TX i RX de la UART

```
void USART1_TX_IRQHandler(void) {
  USART_IntClear( USART1, USART_IEN_TXC);
  USART_Send(USART1);
}

void USART1_RX_IRQHandler(void) {
  char data;

  if (USART1->IF & LEUART_IF_RXDATAV) {
    data = USART_Rx(USART1);
    PushData(data, &RX_SerialBuffer);
    USART_IntClear( USART1, USART_IEN_RXDATAV);
  }
}
```

(**USART_RX()**). Aquestes funcions són molt senzilles, i permeten enviar un sol byte o rebre'n un, o espera-se indefinidament fins que n'arribi un. Això pot ser que no sigui l'ideal per la nostra aplicació i les USART permeten generar interrupcions segons certs esdeveniments. Normalment els més usats són el de byte rebut i el de byte enviat. D'aquesta manera, ens podem muntar una biblioteca que ens permeti rebre dades pel port sèrie fent servir exclusivament interrupcions i a la nostra aplicació agafem el que s'hagi rebut quan vagi bé.

La manera més habitual d'implementar aquest mecanisme asíncron de enviar o rebre dades pel port sèrie és mitjançant un buffer circular. D'aquesta manera, la ISR de recepció (**USART1_RX_IRQHandler()**) insereix la nova dada rebuda al *buffer* de recepció cada cop que la criden. Només cal escriure una funció a la nostra biblioteca que extregui un caràcter del *buffer* cada cop que la cridin i escriure una funció que retorni el nombre de caràcters disponibles al *buffer* (veure Llistat 14.1).

La ISR de transmissió (**USART1_TX_IRQHandler()**) es crida cada cop que s'acaba d'enviar un caràcter, per tant la ISR pot enviar un caràcter del *buffer* circular de transmissió cada cop que la cridin (quan s'ha acabat d'enviar el caràcter anterior) i ens cal una funció que copiï les dades a enviar cap al *buffer* circular de transmissió i engegui el procés.

14.2 Exemple d'ús d'una UART

Per fer les proves caldrà tenir a mà un dispositiu CP2102. Aquest dispositiu conté un xip, el CP2102 que per un costat té un port sèrie i per l'altre un port USB de tipus *Virtual COM Port Device* que qualsevol ordinador, ja sigui Windows, GNU/Linux o Mac reconeixen com un port sèrie. D'aquesta manera, fent servir aquest dispositiu podem connectar un microcontrolador amb un port sèrie a un ordinador de sobretaula o portàtil que tingui un port USB lliure.

Ara el que cal és preparar-ho tot: preparar el codi, connectar el mòdul CP2102 a la nostra placa de prototipat i connectar-la per USB a un ordinador.

El codi que hi ha al repositori primer configura la USART1 del microcontrolador perquè funcioni a 115200 bps, 8 bits per caràcter, sense paritat i 1 bit d'stop, que ve a ser la configuració estàndard d'un port sèrie. També en el mateix bloc configura els dos pins del port (TX i RX). També configura com s'ha d'enrutar la USART1, en aquest cas cap la localització 1 que es correspon als pins PD0 i PD1 del microcontrolador, que estan connectats al connector d'expansió, als pins 4 i 6 respectivament (veure Figura 14.1). Tot seguit s'envia un caràcter 'A' pel port sèrie.

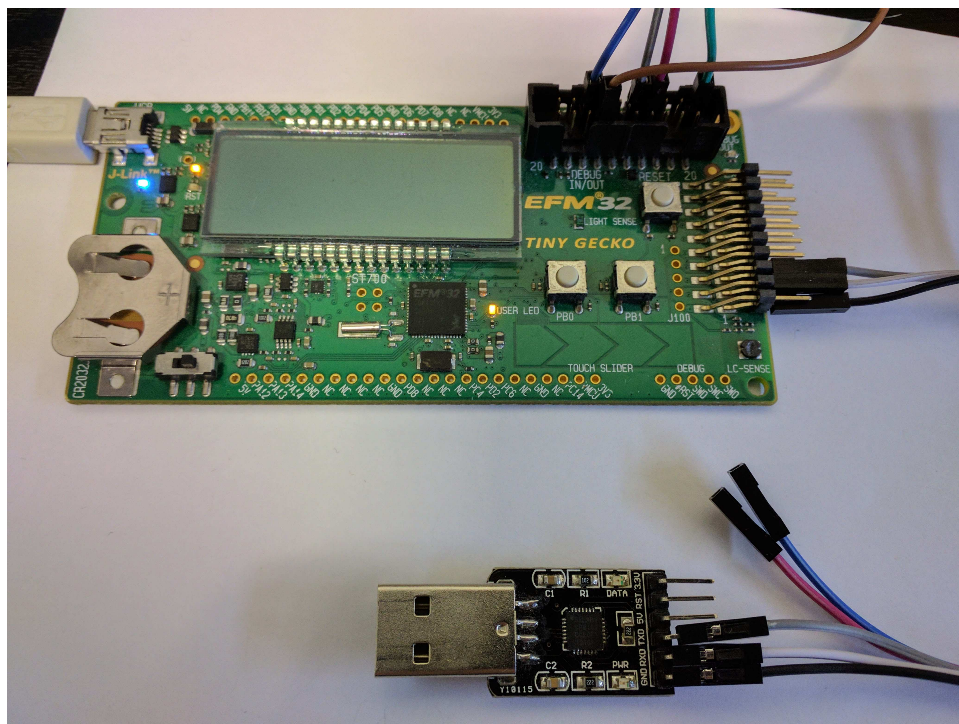


Figura 14.1: Connexió del CP2102 a la placa de prototipat

A continuació es configuren els dos botons perquè generin una IRQ. A les ISRs tant sols es canvia el LED (un botó l'encén i l'altre l'apaga) i un botó envia una 'A' i l'altre botó envia un 'B'.

Per veure aquests caràcters al nostre ordinador, primer cal connectar el mòdul CP2102 a la nostra placa pel connector d'expansió tal com es veu a la foto. Un cop connectem el CP2102 al nostre ordinador, cal engegar algun programa de terminal pel port sèrie, tipus *putty* [37], *minicom*, *Tera Term* [38], etc.

R També es pot instal·lar un terminal dins del **Simplicity Studio**. Cal anar a l'*Eclipse Marketplace*, buscar el **TM Terminal** i instal·lar-lo [39]. Així tindrem el terminal integrat dins el IDE.

14.3 Un exemple amb la UART més complicat

Un cop vist un exemple senzill de com es configura una USART, anem a fer un exemple més complex, rebent i enviant dades del microcontrolador cap al PC i a l'inrevés. El codi està al repositori.

En aquest cas farem servir buffer circulars per guardar les dades rebudes o a enviar. És una implementació senzilla d'un buffer circular sense gaire secrets, l'hem posat a la biblioteca **CircularBuffer**. Aquesta biblioteca ofereix unes funcions molt senzilles per gestionar el buffer circular:

- **PushData()** que permet inserir un caràcter al *buffer*
- **PopData()** que retorna el següent caràcter del *buffer*
- **AvailableData()** que indica la quantitat de caràcters disponibles al *buffer*

En aquest exemple es fan servir les interrupcions de la USART per controlar-la, de manera que al rebre un caràcter, la ISR corresponent l'insereix al *buffer* corresponent (hi ha dos *buffer*, un de

Llistat 14.2: Exemple ISR avançada

```
void USART1_RX_IRQHandler(void) {
  char data;

  if (USART1->IF & LEUART_IF_RXDATAV) {
    data = USART_Rx(USART1);
    PushData(data, &RX_SerialBuffer);
    USART_IntClear( USART1, USART_IEN_RXDATAV);
  }
}
```

Llistat 14.3: Exemple ISR avançada

```
void USART1_TX_IRQHandler(void) {
  USART_IntClear( USART1, USART_IEN_TXC);
  USART_Send(USART1);
}
```

recepció i un d'enviament), veure Llistat 14.2 i 14.3.

D'aquesta manera, en qualsevol moment que es rebi un caràcter per la UART, la ISR el guardarà el *buffer* **RX_SerialBuffer** i seguirà l'execució el nostre programa.

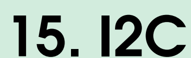
Quan calgui, podrem cridar a la funció **AvailableData()** per saber si hem rebut alguna cosa i processar-la si cal.

Per enviar dades, cal que primer omplim el buffer circular corresponent (**TX_SerialBuffer**) amb el que vulguem enviar i després cridar la funció **USART_Send()**. D'aquesta manera, i gràcies a que hem configurat la USART perquè llenci una IRQ cada cop que acabi d'enviar un caràcter, la USART enviarà un primer caràcter i es cridarà la ISR, que tornarà a cridar la funció d'enviar un caràcter fins que no en quedi cap més al *buffer*.

Fent servir les interrupcions alliberem el processador d'esperar a que la USART vagi enviant els caràcters un a un i ens podem dedicar a fer altres coses (Llistat 14.4).

Llistat 14.4: Funció main

```
while (1) {
  if (AvailableData(&RX_SerialBuffer) != 0) {
    character = PopData(&RX_SerialBuffer);
    /* Prepare the buffer with the data to be sent */
    PushData(character, &TX_SerialBuffer);
    PushData(character + 1, &TX_SerialBuffer);
    PushData(character + 2, &TX_SerialBuffer);
    /* Start send process */
    USART_Send(USART1);
  }
}
```



15. I2C

Quan parlem de busos per comunicar dispositius i sensors amb un microcontrolador, el bus I2C és un dels primers que venen al cap (l'altre és el bus SPI[1], veure **Capítol 16 - SPI**).

Presentat per Philips fa una pila d'anys, ha acabat sent un dels dos busos *estàndard de facto* per comunicació interPCBs (l'altre, com ja hem dit, és l'SPI). És un bus multi-master i multi-slave, que vol dir que poden haver més d'un dispositiu esclau i varis dispositius mestres funcionat al bus (tot i que només es pot comunicar un màster amb un slave alhora). El bus I2C funciona amb només dues línies: una transporta el rellotge del bus (SCL[2]) i l'altra les dades entre tots els dispositius (SDA[3]). Totes dues línies són de tipus *open-drain* i per tant necessita de resistències de *pull-up* pel seu correcte funcionament. La freqüència de funcionament pot ser com a màxim de 100 kHz (o de 400 kHz en mode fast) tot i que pot normalment pot funcionar a qualsevol freqüència inferior.

Cada un dels dispositius té una adreça de 7 bits que ha de ser única al bus i que serveix per adreçar-s'hi per part d'altres dispositius. Una transferència típica consisteix en un Master que posa al bus l'adreça del dispositiu Slave que vol accedir amb el bit menys significatiu a 0 o 1 segons vulgui accedir-hi per llegir ('1') o per escriure ('0'), a continuació posa l'adreça del registre (o adreça de memòria) que vol llegir o escriure i a continuació la dada a escriure o la dada de resposta. La unitat de treball del bus és de 8 bits (1 byte) així que si cal transmetre més dades caldrà enviar les dades byte a byte[4].

Tenim multitud de dispositius de tota mena que tenen aquest bus com sistema de comunicació amb el microcontrolador i és força comú en sensors digitals, memòries EEPROM, expansió d'I/O, ADCs, DACs, etc. i la majoria de microcontroladors inclouen un perifèric tipus Màster.

[1] Serial Peripheral Interface
[2] Serial Clock Line (I2C)
[3] Serial Data (I2C)
[4] Hi ha un mode d'I2C que amplia la longitud de l'adreça de 7 a 10 bits

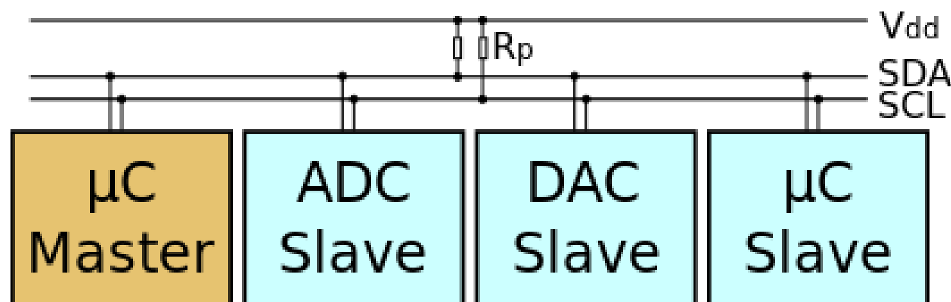


Figura 15.1: Esquema d'un bus I2C típic
Esquema d'un bus I2C típic. Font: user:Cburnett (Own work made with Inkscape) [GFDL o CC-BY-SA-3.0], via la Wikimedia Commons

15.1 Exemple d'I2C

Per aquest exemple farem servir el sensor de gestos, de llum i de proximitat **APDS-9960** de Broadcom [6]. Es pot adquirir una placa de prototipat a la web Sparkfun per poder provar-lo ràpidament. La placa té un connector força estàndard amb els pins d'alimentació, massa, les dues línies del I2C SCL i SDA, i un pin d'interrupció.

Per connectar aquesta PCB a la nostra placa de prototipat, farem servir 4 cables i els connectarem seguint la Taula 15.1.

Amb això estem connectant directament els pins del microcontrolador que poden fer de màster I2C als pins sensor que fa d'slave I2C. Les resistències de pull-up que calen per tot bus I2C estan a la PCB d'sparkfun (veure a l'esquemàtic aquí les resistències R2 i R3).

Un cop tenim el Hardware preparat, cal posar-se amb el Firmware. Per fer servir el controlador Màster del nostre microcontrolador podem fer sevir les llibreries de SiliconLabs emlib. Aquí està el projecte sencer per EFM32.

El primer que cal fer és activar el rellotge per aquest perifèric i configurar els pins corresponents. En aquest cas, els pins 15 o 16 del connector corresponent als pins PD7 i PD6 respectivament, que són la *localization #1* del I2C d'aquest microcontrolador (Llistat 15.1).

A continuació cal configurar el perifèric màster, podem deixar-ho tot amb les opcions per defecte (Llistat 15.2).

Un cop fet això, ja podem fer servir el controlador I2C per accedir als dispositius que hi hagin al bus.

Per comoditat, s'acostuma a munta una funció per llegir registres i una funció per escriure'n. Veiem la funció **I2C_ReadRegister()** (Llistat 15.3).

Taula 15.1: Connexionat de la placa APDS-9960 i la placa de prototipat

SiliconLabs (Expansion header)	APDS-9960 PCB Board
15	SCL
16	SDA
19	GND
20	VDD

Llistat 15.1: Initialització dels pins per l'I2C

```
CMU_ClockEnable(cmuClock_I2C0, true);
GPIO_PinModeSet(gpioPortD, 7, gpioModeWiredAnd, 1);
GPIO_PinModeSet(gpioPortD, 6, gpioModeWiredAnd, 1);

I2C0->ROUTE = I2C_ROUTE_SDAPEN | I2C_ROUTE_SCLPEN |
        I2C_ROUTE_LOCATION_LOC1;
```

Llistat 15.2: Inicialització del perifèric I2C

```
I2C_Init_TypeDef i2cInit = I2C_INIT_DEFAULT;
I2C_Init(I2C0, &i2cInit);
```

Llistat 15.3: Funció **I2C_ReadRegister**

```
static bool I2C_ReadRegister(uint8_t reg, uint8_t *val) {
  I2C_TransferReturn_TypeDef I2C_Status;
  I2C_TransferSeq_TypeDef seq;
  uint8_t data[2];

  seq.addr = DEVICE_ADDR;
  seq.flags = I2C_FLAG_WRITE_READ;
  seq.buf[0].data = &reg;
  seq.buf[0].len = 1;
  seq.buf[1].data = data;
  seq.buf[1].len = 1;

  I2C_Status = I2C_TransferInit(I2C0, &seq);

  while (I2C_Status == i2cTransferInProgress) {
    I2C_Status = I2C_Transfer(I2C0);
  }
  if (I2C_Status != i2cTransferDone) {
    return false;
  }
  *val = data[0];
  return true;
}
```

Llistat 15.4: Funció **testI2C()**

```
void testI2C(void) {
  uint8_t ret_value;

  I2C_ReadRegister(0x92, &ret_value);

  if (ret_value == 0xAB) {
    printf("Detected!\r\n");
  } else {
    printf("Something went wrong\r\n");
  }
}
```

El primer que fem és definir les variables que ens caldran. La variable més estranya que veiem aquí és la variable **seq** de tipus **I2C_TransferSeq_TypeDef**. En aquesta estructura és on es guarda tota la informació necessària per engegar una transacció per part del controlador dins el microcontrolador. Per tant, a l'estructura cal posar-hi l'adreça I2C del dispositiu, quina mena de transacció es vol fer (lectura, escriptura, etc.), en aquest cas **I2C_FLAG_WRITE_READ**, que vol dir que volem llegir d'un registre; també cal omplir una *array* d'estructures (**buf[]**) on es guarda les dades a enviar o a rebre. En aquest cas, com que només volem llegir un registre (un sol byte) al **buf[0]** només hi posem l'adreça del registre a llegir i la longitud a 1 i a **buf[1]** li posem un *buffer* per emmagatzemar el valor llegit i de longitud 1 sol byte per llegir.

Un cop omplerta aquesta estructura s'engega la transacció amb la crida **I2C_TransferInit()**, que retorna un **I2C_Status**. Després, tal com ens demana la biblioteca EMLIB, cal anar cridant la funció **I2C_Transfer()** mentre la variable **I2C_Status** valgui **i2cTransferInProgress**. Això indica que la transferència pel bus s'ha acabat i cal veure com ha anat l'operació. Si tot ha anat bé la variable **I2C_Status** valdrà **i2cTransferDone**. Així, el que tenim és una funció que li passem l'adreça del registre que volem llegir i ens retorna el valor que ens retorna el dispositiu.

Ara tornem al nostre sensor de gestos. Quasi tots els dispositius I2C tenen un registre de tipus identificació, que ens permet validar que la comunicació es fa realment amb ell. En el cas del APDS-9960 té un registre anomenat 'ID' a l'adreça 0x92 i que el valor que retorna al llegir-lo és 0xAB (pàgina 25 del *Datasheet* [6]). Ara toca, doncs, muntar una funció que comprovi que s'està correctament a aquest dispositiu com es veu al Llistat 15.4.

Ara només cal ajuntar-ho tot en el nostre main i ja podem comprovar que tenim la PCB ben connectada i funcionant. Farem servir aquest bus i el mateix dispositiu a l'exemple tractat a **Capítol 21 - Una aplicació completa**.

16. SPI

El bus SPI és un altre dels busos més populars usats en sistemes encastats. Aquest sistema de comunicació no és un bus pròpiament dit, donat que ens permet connectar un *Master* amb un *Slave* i només de forma indirecta connectar més *Slaves*. Aquest bus es basa en dues línies per enviar o rebre dades, batejades MOSI[1] i MISO[2] segons la dada vagi del *Master* cap a l'*Slave* o a l'inrevés. Com és un bus síncron, hi ha una tercera línia que porta el rellotge (SCLK[3]) i que el genera el *master*. Per últim, hi ha una quarta línia de control, normalment anomenada SS[4] que indica a un *slave* concret que la transmissió és per ell. Controlant correctament un conjunt d'aquestes línies SS és possible tenir més dispositius *slaves* al connectats al bus (Figura 16.1). Aquesta línia pot no usar-se, i llavors es parla d'un SPI de 3 fils enlloc d'un SPI de 4 fils.

Aquest bus acostuma a poder treballar a força més velocitat que el bus I2C, i és habitual configurar-lo per freqüències de rellotge d'uns pocs MHz (és habitual treballar amb freqüències d'1, 10 o fins i tot 24 MHz). En tot cas, cal consultar sempre les especificacions dels dispositius *slave* per saber la velocitat màxima de treball.

Precisament per aquesta característica, que fa que es puguin enviar moltes més dades per segon, els dispositius que incorporen un bus SPI acostumen a ser sensor i dispositius que generen un gran volum de dades de forma continua, com ara: ADCs, memòries RAM i FLASH, LCDs, targetes SD, alguns sensors concrets, etc.

Com que aquest bus no està pensat per tenir més d'un *slave* connectat simultàniament, no incorpora el concepte d'adreça de dispositiu. A diferència del I2C, que el propi bus especifica com és el protocol per accedir a un registre, en el SPI aquesta definició és pròpia de cada fabricant. Tot i això, habitualment el primer camp de la transmissió és l'adreça del registre a accedir o la comanda a executar.

[1] *Master Output Slave Input*, Sortida del Master Entrada a l'Esclau
[2] *Master Input Slave Output*, Entrada al Master Sortida de l'Esclau
[3] *Serial Clock*, Relloge Master
[4] *Slave Select*, Selecció d'Esclau

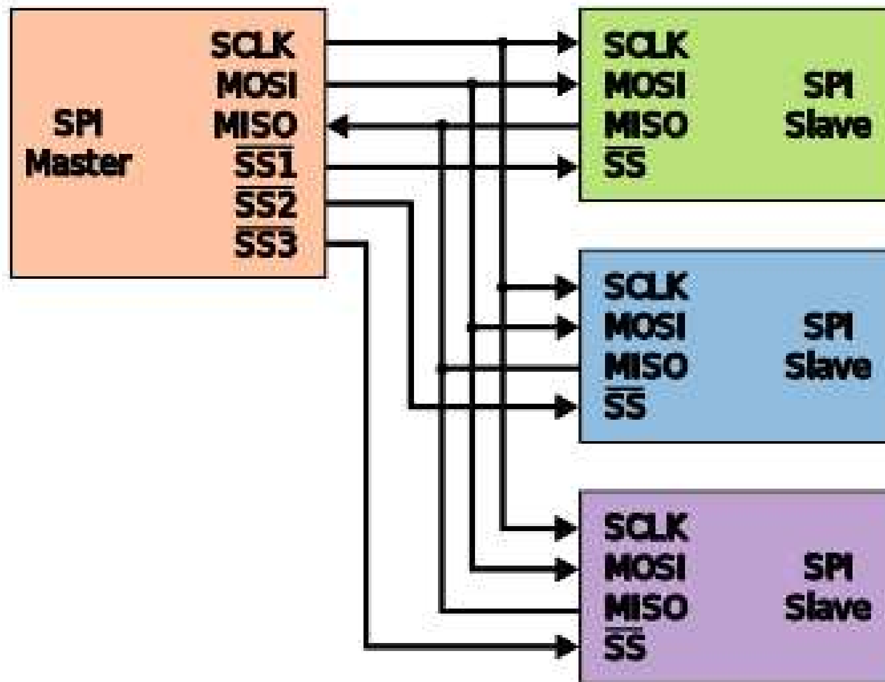


Figura 16.1: Bus SPI típic
Bus SPI típic. Per en:User:Cburnett [GFDL (http://www.gnu.org/copyleft/fdl.html) o CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/)], de la Wikimedia Commons

En alguns microcontroladors el control d'aquest bus el fa el perifèric USART (veure **Capítol 14 - UART**) configurat pertinentment, com és el cas de les famílies de microcontroladors de Silicon Labs [24, pàgina 207], i d'altres famílies incorporen perifèrics dedicats en exclusiva a aquest bus, com els de ST [30, pàgina 873].

17. DMA

El DMA[1] és un dispositiu present a la majoria de microprocessadors. Aquest dispositiu s'encarrega de fer transferències de dades entre dispositius o memòria sense que la CPU intervingui.

Per què es fa servir? Doncs perquè sovint cal transferir quantitats importants de dades entre dues parts de la memòria, o anar transferint dades d'un dispositiu cap a un *buffer* a la memòria i no cal que la CPU estigui entretinguda fent això quan podria fer alguna cosa més interessant o en un mode de baix consum (veure **Capítol 35 - Baix cosum**).

Els DMA acostumen a tenir diferents canals que poden fer transferències per separat i en paral·lel, de manera que podem tenir un canal preparat per moure regions de memòria, un altre per enviar dades cap a un DAC, un tercer transferint les dades rebudes per la UART, etc. Aquest diferents canals acostumen a estar prioritzats, de manera que l'àrbitre del bus pot decidir quin canal ha de poder accedir abans que un altre.

Normalment el dispositiu DMA es configura mitjançant descriptors, que són unes estructures on es guarda el tipus de transferència, l'adreça d'origen, l'adreça de destí, etc. Un cop s'engega la transferència, el DMA anirà movent les dades d'una adreça a l'altra segons s'hagi configurat. A la majoria de DMAs moderns, a més, es pot configurar el DMA de manera que copiï una dada d'un determinada adreça (que correspon a un registre de dades d'un perifèric) cada cop que el perifèric llença una notificació, de manera que el DMA fa una còpia d'una dada cada cop que el dispositiu en genera una. Tot això es veurà millor amb dos exemples.

17.1 Exemple

Posar un exemple de DMA sovint és complicat, ja que involucra força perifèrics diferents i configuracions, així que intentarem fer algun de simple per començar. Per aquest primer exemple hem optat per fer una transferència entre dues regions de memòria amb el DMA. Per això, tindrem

[1] *Direct Memory Access*

Llistat 17.1: Inicialització del DMA

```
dma_init.hprot = 0;
dma_init.controlBlock = dmaControlBlock;
DMA_Init(&dma_init);
```

Llistat 17.2: Definició de la variable **dmaControlBlock**

```
DMA_DESCRIPTOR_TypeDef dmaControlBlock[DMACTRL_CH_CNT * 2] SL_ATTRIBUTE_ALIGN(
    DMACTRL_ALIGNMENT);
```

un buffer d'origen i un de destí, en aquest cas de 256 elements de 32 bits cadascun (uint32_t src_buffer[BUFFER_SIZE]).

El primer que es fa és inicialitzar el DMA amb les funcions d'emlib corresponents.

La variable **dmaControlBlock** és un *array* on es guarden els descriptors de cada un dels canals disponibles del DMA (Llistat 17.2).

A continuació configurem un canal del DMA (Llistat 17.3) i li assignem una funció de callback perquè ens notifiqui quan ha acabat la transferència (Llistat 17.4):

També s'activen les interrupcions (necessàries per tenir callbacks) i es selecciona com serà la transferència (en aquest cas un 0 per indicar que serà de memòria a memòria).

Per últim es configura el descriptor de la transferència (Llistat 17.5).

En aquest cas es tria com ha de manegar les adreces (tant a la font com al destí) i quants bytes es transfereixen a cada accés. Com que estem movent un *array* de 32 bits de paraula, triem incrementar de 4 en 4 l'adreça i moure 4 bytes a cada accés.

Per últim activem la transferència DMA com es veu al Llistat 17.6.

Aquí la única particularitat és que cal especificar-li el nombre d'accessos a fer per completar la transferència, però restant-li un[2] i altres biblioteques poden necessitar paràmetres lleugerament diferents.

A l'exemple es fa servir una variable global (*volatile*!) de nom **dma_end** que la callback posa a

[2]Això és una particularitat de la biblioteca EMLIB de *Silicon Labs*

Llistat 17.3: Configuració del canal DMA

```
dma_cb.cbFunc = DmaComplete;
dma_cb.userPtr = NULL;

dma_chn.enableInt = true;
dma_chn.highPri = false;
dma_chn.select = 0; /* set to 0 because is a memory-to-memory transfer */
dma_chn.cb = &dma_cb;
DMA_CfgChannel(0, &dma_chn);
```

Llistat 17.4: Callback del DMA **DmaComplete()**

```
void DmaComplete(unsigned int channel, bool primary, void *user) {
  dma_end = true;
}
```

Llistat 17.5: Paràmetres de configuració del DMA

```
dma_descr.arbRate = dmaArbitrate1;
dma_descr.dstInc = dmaDataInc4;
dma_descr.hprot = 0;
dma_descr.size = dmaDataSize4;
dma_descr.srcInc = dmaDataInc4;
DMA_CfgDescr(0, true, &dma_descr);
```

true per notificar a la funció **main()** que la transferència s'ha acabat. Quan això passa, es comprova que els dos *buffers* són iguals (Llistat 17.7) i es presenta per la consola.

Llistat 17.6: Activació de la transferència DMA

```
DMA_ActivateAuto(0, true, dst_buffer, src_buffer, BUFFER_SIZE - 1);
```

Llistat 17.7: Comparació dels dos buffers de l'exemple DMA

```
bool check_buffers_copy() {
  int i;

  for (i=0; i<BUFFER_SIZE; i++) {
    if (dst_buffer[i] != src_buffer[i]) {
      return false;
    }
  }
  return true;
}
```

17.2 Un exemple amb DMA més complicat

Una altra aplicació del DMA és transferir dades cap o des de un perifèric cap a memòria sense que el processador ho hagi de fer. Un exemple és fer servir el DMA per rebre o transferir dades cap al port sèrie. Així, enlloc d'anar llençant una IRQ cada cop que el microcontrolador pot enviar la següent dada, es pot configurar el DMA per a que ho faci ell mateix de forma automàtica Anem a veure com es fa això modificant el segon exemple sobre la UART (comentat a la **Secció 14.3 - Un exemple amb la UART més complicat**).

Així, el que farem en aquest exemple és deixar el buffer circular i el mateix mecanisme per rebre dades de la **USART (RX)** però canviarem la manera en que enviem les dades (TX). L'exemple és aquí.

El que farem és preparar el DMA per a que transfereixi dades des d'un *buffer* a la memòria cap al registre d'enviament de la USART (registre **TXDATA** de la USART1). En aquest cas, cal configurar-lo que no incrementi l'adreça de destí, que incrementi la font byte a byte i que transfereixi bytes (els caràcters d'una UART són bytes). També cal especificar quin dispositiu fa de *trigger*, és a dir, quin dispositiu notifica que té una dada disponible per ser transferida (paràmetre dma_chn.select = DMAREQ_USART1_TXBL). En aquest exemple no ens cal saber quan el DMA ha acabat la transferència, així que no registrem cal callback ni activem les interrupcions del DMA. Tot això ho hem encapsulat a la funció **my_DMA_Init()** (Llistat 17.8).

Ens cal també una funció que engegui la transferència del DMA cap a la UART. Li hem dit **sendUARTbyDMA()** i rep com a paràmetres el *buffer* a enviar i la seva longitud (en bytes). Aquesta funció tan sols espera que el DMA no estigui ocupat i tot seguit engega la transferència (veure Llistat 17.9).

Llistat 17.8: Diferències a la inicialització del DMA

```
void my_DMA_Init(void) {
  ...
  dma_chn.select = DMAREQ_USART1_TXBL;
  ...
  dma_descr.dstInc = dmaDataIncNone;
  dma_descr.size = dmaDataSize1;
  dma_descr.srcInc = dmaDataInc1;
  ...
}
```

Llistat 17.9: Funció sendUARTbyDMA()

```
void sendUARTbyDMA(void *buffer, int size) {
  /* wait for other DMA transfer to complete */
  while (DMA_ChannelEnabled(DMA_USART_TX_CHANNEL));

  /* Activate DMA */
  DMA_ActivateBasic(DMA_USART_TX_CHANNEL, true, false, (void *) &(USART1->
      TXDATA), buffer, size - 1);
}
```

Llistat 17.10: Funció **main()** de l'exemple

```
void main() {
  ...
  while (1) {
    if (AvailableData(&RX_SerialBuffer) != 0) {
      character = PopData(&RX_SerialBuffer);
      sprintf(TXbuffer, "Send by DMA (%c)\r\n", character);
      sendUARTbyDMA(TXbuffer, strlen(TXbuffer));
    }
  }
  ...
}
```

La resta del codi és prou auto-explicatiu (o ho hauria de ser!): no s'activa la IRQ de TX de la USART i hem tret la ISR corresponent. Tampoc es crea un buffer circular per enviar per la UART.

Dins el bucle infinit del **main()** s'espera a rebre un caràcter, es munta una cadena relativament llarga (17 bytes) amb el caràcter rebut i per acabar s'envia usant la nova funció.

Per veure com connectar i fer servir el port sèrie per comunicar-se amb un ordinador, mireu el **Capítol 14 - UART**.

Aquesta pàgina està en blanc expressament, tot va bé.

18. FLASH

La memòria FLASH que actua com a ROM pel nostre microcontrolador també es pot fer servir per emmagatzemar dades d'usuari i ser escrita i llegida per codi de programa. Com que aquesta mena de memòria és no volàtil, les dades que hi guardem romandran entre reinici o pèrdues d'alimentació.

Cal tenir en compte que les memòries FLASH tenen certes particularitats que les fan diferents a d'altre tipus de memòries:

- Número finit de cicles d'escriptura i lectura: les memòries FLASH tenen un límit de vegades que es poden esborrar i escriure. Aquest nombre pot estar entre 10.000 i 100.000 cicles, cosa que les pot fer poc adequades per emmagatzemar dades que variïn molt sovint.
- Paginació: les memòries FLASH estan organitzades en pàgines que cal esborrar sencera per escriure una dada.
- No es pot llegir mentre s'escriu la memòria FLASH del microcontrolador, per tant caldrà tenir una cura especial en que no hi hagi cap lectura ni execució de codi (compte amb les ISRs) mentre s'està escrivint dades a la FLASH.

En el cas dels microcontroladors de Silicon Labs, el nombre màxim d'escriptures a la FLASH és de 20.000 cicles, un cop passada aquesta xifra la FLASH pot començar a donar problemes [40, pàgina 28] (en els microcontroladors d'ST, el nombre de cicles garantits és de 10.000 [41, pàgina 110]). Això fa que, per exemple, si volem emmagatzemar les dades d'un sensor que es llegeixen cada 10 segons a la FLASH, estaríem superant les 20.000 escriptures al cap de 55 dies de funcionament. Caldrà buscar mètodes alternatius d'emmagatzemar les dades per tal de no fer malbé la FLASH tant aviat.

18.1 Un exemple senzill

A continuació veurem un exemple senzill on guardarem un conjunt de dades que han de ser persistents entre reinicis o pèrdues d'alimentació del nostre dispositiu. L'exemple està al repositori amb el nom de FLASH_1.

Llistat 18.1: Estructura per guardar-se a la FLASH

```
typedef struct {
  uint32_t field1;
  uint8_t  field2;
  uint32_t field3;
  bool     field4;
} persistent_data_t;
```

Llistat 18.2: Funcions per accedir a la FLASH

```
void Flash_Write() {
  MSC_ErasePage(userDataPage);
  MSC_WriteWord(userDataPage, &my_persistent_data, sizeof(my_persistent_data));
  printf("Data stored in Flash\r\n");
}

void Flash_Read() {
  memcpy(&my_persistent_data, userDataPage, sizeof(my_persistent_data));
}
```

Primer es defineix una estructura en C per agrupar les dades que es volen emmagatzemar a la FLASH tal com es veu al Llistat 18.1. Tots els camps d'aquesta estructura seran les dades persistents que guardarem a la FLASH i que es podran llegir de nou a cada reinici.

Tot seguit es defineixen dues funcions per llegir i escriure a la FLASH (Llistat 18.2). Com es pot veure, cal cridar a funcions especials per escriure a la FLASH: la primera (**MSC_ErasePage()**) esborra una pàgina de la FLASH i la segona (**MSC_WriteWord()**) escriu a la FLASH les dades que se li passin. Aquesta funció d'escriptura necessita que la mida de les dades sigui divisible per 4 (en el cas de la nostra estructura, la seva mida és de 16 bytes) [42][43].

Com es pot veure al mateix codi (Llistat 18.2), per llegir de la memòria FLASH (funció **Flash_Read()**) només cal fer un accés a memòria normal i corrent, i en aquest cas el que es fa és copiar el contingut de la FLASH cap a la estructura amb les dades persistents amb un **memcpy()**.

La resta de l'exemple és prou senzill: quan es prem el botó 0 de la placa d'avaluació s'escriu a la FLASH l'estructura i al pitjar el botó 1 es canvien els valors de dita estructura. D'aquesta manera, al iniciar la sessió de *debug* es pot observar com els valors que hi han guardat a la FLASH i que es copien a l'estructura són els valors que s'han guardat prèviament a l'anterior execució.

Per últim, cal fer notar que la macro **USERDATA_BASE** està definida pel fabricant i apunta a la zona de la FLASH reservada per l'usuari d'una pàgina (en aquest cas 512 bytes) de longitud.

18.2 *Bootloaders*

Un cas especial per accedir a la FLASH és el dels *bootloaders*. Un *bootloader* és un petit codi que s'executa cada cop que s'inicia el microcontrolador i acostuma a poder gestionar la reprogramació del microcontrolador d'una forma més amigable que no pas usant el programador.

El més habitual en *bootloaders* per sistemes encastats és que puguin rebre una nova imatge de

l'executable de l'aplicació a través d'un dels ports sèrie del sistema [1]. Per realitzar aquest tasca, el *bootloader* ha de poder accedir i escriure a tota la memòria FLASH del microcontrolador [44].

[1]També hi ha *bootloaders* que poden rebre la imatge per USB, via ràdio, per un port Ethernet, llegir-la d'una tarja SD, etc.

Aquesta pàgina està en blanc expressament, tot va bé.

19. Mòduls criptogràfics

Molts dels microcontroladors actuals incorporen perifèrics per accelerar els càlculs de xifratge i desxifrage de dades. La majoria suporten directament el xifratge i desxifratge dels mètodes més habituals (AES, DES, 3DES, etc.) i proporcionen acceleració a d'altres mètodes més inusuals. A més, acostumen anar acompanyats de biblioteques que ajudes a un ús senzill d'aquests processos que, en ocasions, son força complexos.

Pel cas de la família EFM32 es te un mòdul criptogràfic compatible amb AES en les versions més antigues dels microcontroladors i un mòdul millorat anomenat **CRYPTO** a les famílies més modernes [4, pàgina 453]. Pel microcontroladors d'ST hi ha un perifèric anomenat processador criptogràfic (*Cryptographic processor*) i el *Hash processor* per realitzar tasques relacionades amb el xifratge de dades [30, pàgina 720].

En el cas de SiliconLabs i el microcontrolador que tenim a la nostra placa de prototipat (Tiny Gecko), el mòdul AES suporta, com el seu propi nom indica, el mètode de xifratge AES, en les versions de clau de 128, 192 i 256 bits i treballant en blocs de 128 bits de dades. Xifrar un missatge de 128 bits li porta 54 cicles de rellotge en el cas d'una clau de 128 bits i de 75 cicles amb la clau de 256 bits. És per tant, una implementació força ràpida i eficient de l'algorisme i, no cal dir-ho, millor i més robusta que la que puguem fer nosaltres amb codi.

La versió del mòdul per microcontroladors més moders, anomenada CRYPTO, accelera també funcions de hash (SHA-1, SHA-224 i SHA-256). parts de xifratge el·líptic (ECC) i porta acompanyant una biblioteca SW que suporta d'altres algorismes (DES, 3DES, MD5, RC4) accelerats en part pel mòdul HW.

L'ús d'aquesta mena de mòduls acostuma a ser força senzill, ja que només cal configurar la clau de xifratge i el mètode a usar i a partir d'aquí emplenar el buffer i donar l'ordre de xifrar (o desxifrar). Un cop acabat el xifratge, es pot llegir el buffer amb les dades xifrades i fer-les servir com calgui.

Llistat 19.1: Clau i text a xifrar

```
uint8_t myKey[16] = {0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2A, 0x2B, 0x2C
    , 0x2D, 0x2E, 0x2F, 0x30, 0x31, 0x32};
char my_message[] = "This is a plain message to be encrypted with the AES
    module.";
uint8_t my_buffer[64];
```

Llistat 19.2: Operació de xifratge

```
AES_ECB128(my_buffer, (uint8_t*) my_message, my_message_len, myKey, true);
```

19.1 Xifrant dades amb AES-128

Al repositori hi ha un exemple que xifra una cadena de text amb AES-128 i el mètode conegut com ECB (Electronic Codebook), el mètode més senzill de xifratge. Primer cal generar una clau de xifratge i un text a xifrar (Llistat 19.1).

Donat que la biblioteca emlib suporta el perifèric, només cal una crida a la funció especifica per obtenir el xifrat del buffer, tal com es veu al Llistat 19.3.

Un cop retorna la funció, a *my_buffer* ja hi tindrem el missatge xifrat. A l'exemple, agafem aquest buffer xifrat i el desxifrem per comprovar que, efectivament, el procés ha estat correcte:

Aquí cal fer notar que primer hem de generar la clau de desxifratge a partir de la clau de xifratge. Aquest procés també es fa via HW amb el perifèric AES.

A la consola es van treien els valors que es van obtenint a cada pas, tal com es veu a la Figura 19.1.

El resultat també el podem comprovar a una eina externa per corroborar que el procés és vàlid i compatible amb algun altre SW de xifratge/desxifratge AES-128. Podem fer servir la web *aes online domain tools* per comprovar-ho (link, veure Figura 19.2).

Llistat 19.3: Operació de xifratge

```
/* Generate decrypt key from original key */
AES_DecryptKey128(decryptionKey, myKey);

/* Decrypt message */
AES_ECB128((uint8_t) my_buffer_decrypt, my_buffer, my_message_len,
    decryptionKey, false);
```

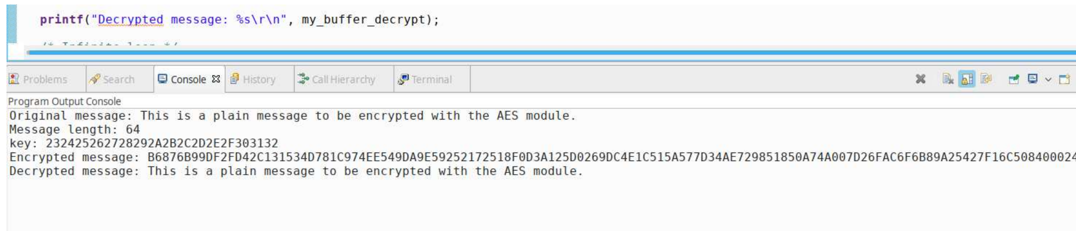


Figura 19.1: Consola de l'exemple AES_1

AES – Symmetric Ciphers Online

Input type: Text

Input text: (hex)
B6876B99DF2FD42C131534D781C974EE549DA9E59252172518F0D3A125D0269DC4E1C515A577D
34AE729851850A74A000B31B814A2656BD198860A78DF97A367

Plaintext Hex Autodetect: ON | OFF

Function: AES

Mode: ECB (electronic codebook)

Key: (hex) 232425262728292A2B2C2D2E2F303132

Plaintext Hex

> Encrypt! > Decrypt!

Decrypted text:

```
00000000  54 68 69 73 20 69 73 20 61 20 70 6c 61 69 6e 20  This is a plain
00000010  6d 65 73 73 61 67 65 20 74 6f 20 62 65 20 65 6e  message to be en
00000020  63 72 79 70 74 65 64 20 77 69 74 68 20 74 68 65  crypted with the
00000030  20 41 45 53 20 6d 6f 64 75 6c 65 00 58 00 00 00   AES module.X...
```

[Download as a binary file] [?] Inactive

Figura 19.2: Web per desxifrar el text xifrat de l'exemple AES_1

Aquesta pàgina està en blanc expressament, tot va bé.

20. Altres perifèrics

Fins ara s'han introduït els perifèrics més habituals i que es poden trobar a als microcontroladors actuals. Tot i això, hi ha altres perifèrics més específics d'algun àmbit d'aplicació que no s'han presentat. En podem enumerar uns quants sense poder ser exhaustius:

- DMA avançats, que permeten fer transferències complexes, no tant sols d'un *buffer* d'una sola dimensió (com els *arrays*) si no de dues dimensions per moure imatges o matrius de 2 dimensions [30, pàgina 339].
- *Drivers* LCD[1] que poden controlar directament pantalles [30, pàgina 480][4, pàgina 490]. Segons el model i el fabricant aquest mòdul podrà controlar diferents tipus de pantalla (LCD o TFT), amb color o blanc i negre, diferents resolucions i profunditat de color, etc.
- Mòdul controlador de USB que permet connectar i controlar el microcontrolador al bus USB [30, pàgina 965][45, pàgina 1446].
- Mòdul Ethernet per poder connectar el nostre sistema encastat a una xarxa cablejada amb Ethernet. Aquests tipus de perifèrics necessiten un xip extern que es controla per un bus força estandarditzat anomenat MII[2] (o el més modern RMII) [30, pàgina 1121][45, pàgina 1729] [46].
- Controlador SDIO[3] per accedir a dispositius SDIO, majoritàriament targetes de memòria SD [30, pàgina 1019][45, pàgina 1670].
- Controlador de bus CAN[4] que permet connectar el dispositiu al bus CAN [30, pàgina 1076][45, pàgina 1899] [47].

Treballar amb aquests perifèrics, tot i que no ho veurem en aquest llibre, és força similar a la resta de perifèrics que hem vist: estan mapats a memòria i tenen un conjunt de registres que permet controlar el perifèric a través de llegir i escriure aquests registres. Els fabricants proporcionen llibreries per facilitar-ne l'ús com amb la resta de perifèrics que hem vist.

[1] *Liquid Crystal Display* Pantalla de Cristall Líquid
[2] *Media-independent interface*
[3] *Secure Digital Input Output*
[4] *Controller area network*

Els perifèrics més complexos (per exemple USB o Ethernet) acostumen a portar associats una llibreria proporcionada pel fabricant o per tercers per poder controlar el perifèric i simplificar-ne l'ús. Així, per exemple, és habitual fer servir la llibreria LwIP per tenir funcionalitat de xarxa (*sockets*, TCP, UDP, IP, etc.) [48][49] i que els fabricants proporcionin documentació o codi per enllaçar aquesta llibreria amb el seu perifèric.

20.1 *Peripheral Reflex System*

Un perifèric que proporciona el fabricant Silicon Labs per la seva línia EFM32 és el *Peripheral Reflex System* (PRS[5]). Aquest sistema és una mena de xarxa que permet a diferents perifèrics comunicar-se entre si sense involucrar la CPU, de manera que uns envien senyals que d'altres recullen per engegar alguna tasca [4, pàgina 135].

D'aquesta manera, és possible que un Timer enviï un senyal a la UART perquè inici una transmissió, o un GPIO engegui un Timer i permeti comptar quant de temps ha estat pitjat un botó. Tot això es fa sense que la CPU intervingui per a res, estalviant energia i simplificant el codi. Els senyals que es generen i es reben estan dins de canals, de manera que poden funcionar diferents canals simultàniament i connectar-hi diferents perifèrics sense que s'interfereixin. Aquest perifèric només es troba als dispositius de Silicon Labs i, fins al moment, cap altre fabricant inclou res de similar en els seus microcontroladors [50].

20.1.1 Un exemple amb PRS senzill

En aquest exemple d'ús del PRS el farem servir per comptar el temps que es polsa un dels botons de la placa de prototipat. Per fer-ho, configurarem el port GPIO per a que generi un senyal PRS per nivell i aquest alimenti a un dels canals del Timer 0. Quan es detecti el flanc de baixada del GPIO (es pitja el botó), el Timer començarà a comptar i quan es detecti el flanc de pujada (es deixa anar el botó) el Timer s'aturarà i llençarà una interrupció de tipus *Input Capture*. Al final, el que tindrem al registre *Capture* 0 del Timer serà el nombre de *ticks* que ha estat pitjat el botó 0.

La configuració del Timer és la que es veu al Llistat 20.1. Al codi es veu que es tria el canal 2 del PRS per activar l'*input capture* i es configura el Timer per a que s'engegui quan es rebi un flanc de baixada i s'aturi amb el flanc de pujada. També es pre-escala el rellotge per 1024, deixant-lo en $14.000.000/1024 = 13.617,875Hz$.

La configuració del pin és més senzilla, tal com es veu al Llistat 20.2. Simplement es configura que els GPIO generaran senyals PRS i es configura el canal número 2 per que rebi el valor del pin 8.

Per últim, el Timer està configurat per generar una IRQ en quan capturi el valor del comptador el CC1. Per tant, cal escriure la ISR corresponent i tractar les dades com toca. En aquest cas només es treu per la consola el valor llegit en ticks del Timer. A la funció **main()** tant sols hi ha la configuració dels perifèrics, ja que un cop configurats, la CPU no cal que faci res fins que no es crida la ISR; és per això que dins el bucle principal es posa la CPU en el mode EM1 de baix consum amb la crida a la funció **EMU_EnterEM1()**.

20.1.2 Exemple amb PRS, DMA, DAC i ADC

Anem a veure un exemple força complex, on intervindran el DMA, l'ADC, el DAC i un parell de Timers. El que farà l'exemple serà enregistrar els valors durant 2 segons d'una entrada analògica per replicar-la després per una sortida també analògica. Per això, es configurarà l'ADC per que faci les lectures i es vagin guardant a un buffer fent servir el DMA i després es faci l'operació a la inversa cap el DAC.

[5] *Peripheral Reflex System*

Llistat 20.1: Configuració del Timer i l'*input capture*

```
static void TimerConfig(void) {
  TIMER_InitCC_TypeDef timerCCInit = {
    .eventCtrl = timerEventFalling,
    .edge = timerEdgeFalling,
    .prsSel = timerPRSSELCh2,
    .cufoa = timerOutputActionNone,
    .cofoa = timerOutputActionNone,
    .cmoa = timerOutputActionNone,
    .mode = timerCCModeCapture,
    .filter = false,
    .prsInput = true,
    .coist = false,
    .outInvert = false
  };

  TIMER_InitCC(TIMER0, 0, &timerCCInit);

  TIMER_Init_TypeDef timerInit = {
    .enable = false,
    .debugRun = false,
    .prescale = timerPrescale1024,
    .clkSel = timerClkSelHFPerClk,
    .fallAction = timerInputActionReloadStart,
    .riseAction = timerInputActionStop,
    .mode = timerModeUp,
    .dmaClrAct = false,
    .quadModeX4 = false,
    .oneShot = false,
    .sync = false
  };

  TIMER_Init(TIMER0, &timerInit);
  TIMER_IntEnable(TIMER0, TIMER_IF_CC0);
  NVIC_EnableIRQ(TIMER0_IRQn);
}
```

Llistat 20.2: Configuració del GPIO per generar un senyal PRS

```
 static void GPIOConfig(void) {
  GPIO_PinModeSet(gpioPortD, 7, gpioModePushPullDrive, 0); /* LED */
  GPIO_PinModeSet(gpioPortD, 8, gpioModeInput, 0);            /* Boto 0 */
  GPIO_PinModeSet(gpioPortB, 11, gpioModeInput, 0);           /* Boto 1 */

  /* Set Interrupt configuration for both buttons */
  GPIO_IntConfig(gpioPortD,  8, false, true, true);
  GPIO_IntConfig(gpioPortB, 11, false, true, true);

  GPIO_InputSenseSet(GPIO_INSENSE_PRS, _GPIO_INSENSE_RESETVALUE);

  PRS_SourceSignalSet(2, PRS_CH_CTRL_SOURCESEL_GPIOH,
      PRS_CH_CTRL_SIGSEL_GPIOPIN8, prsEdgeOff);
}
```

Llistat 20.3: ISR del Timer

```
void TIMER0_IRQHandler(void) {
  volatile uint32_t time_value = 0;

  uint32_t aux;
  aux = TIMER_IntGet(TIMER0);

  TIMER_IntClear(TIMER0, aux);

  time_value = TIMER_CaptureGet(TIMER0, 0);
  printf("time button 0: %lu\r\n", time_value); // 13672 ticks / second
}
```

Llistat 20.4: Configuració de l'ADC perquè funcioni amb el PRS

```
static void ADCConfig(void) {
  ...
  singleInit.reference = adcRefVDD;
  singleInit.input = adcSingleInpCh6;

  /* Use PRS channel 0 */
  singleInit.prsEnable = true;
  singleInit.prsSel = adcPRSSELCh0;
  ADC_InitSingle(ADC0, &singleInit);
  ...
}
```

Per marcar el ritme de captura de l'ADC i de conversió del DAC es farà servir un senyal PRS proporcionat per Timers. D'aquesta manera, es configurarà un *Timer* per a que generi un senyal PRS cada cop que fa *overflow* i aquest senyal engegui el procés de conversió de l'ADC i un altre Timer per generar un senyal similar pel DAC.

A l'exemple es configurarà l'ADC perquè prengui mostres del canal 6 amb referencia de tota l'escala, tot seguit es configura perquè el seu *trigger* sigui el canal 0 del PRS (veure Llistat 20.4).

A continuació es configura el canal PRS perquè sigui el Timer 0 qui generi el senyal i es configurà també el Timer 0 perquè generi un pols a la freqüència desitjada (Llistat 20.5).

Amb això tindrem que quan s'engegui el Timer 0, aquest anirà generant senyals pel PRS que faran que l'ADC faci una conversió de senyal. Ara cal configurar el DMA perquè reculli aquesta dada i l'emmagatzemi on pertoca. Això es farà de forma molt similar a l'exemple anterior, configurant el canal de manera que la font del senyal serà l'ADC (paràmetre DMAREQ_ADC0_SINGLE) i que l'origen de dades no s'ha de canviar i el destí s'ha d'incrementar de 2 en 2 (ja que llegirem dades de l'ADC de tipus uint16_t que son de 2 bytes) (Llistat 20.6).

Per últim, per engegar aquest procés ho farem a través del botó 0 de la placa i per tant hem de posar el codi que engega el Timer 0 i que activa el DMA a la ISR corresponent (Llistat 20.7).

Un procés molt similar és el que cal fer per realitzar l'operació inversa, és a dir, fer que el DMA transfereixi les dades obtingudes cap al DAC perquè aquest generi el senyal analògic corresponent. Primer es configura el DAC indicant que cal que faci una conversió cada cop que rebi un senyal del canal PRS número 3 (Llistat 20.8).

Llistat 20.5: Configuració del Timer0 perquè funcioni amb el PRS

```
static void ADCConfig(void) {
  ...
  PRS_SourceSignalSet(0, PRS_CH_CTRL_SOURCESEL_TIMER0,
      PRS_CH_CTRL_SIGSEL_TIMER0OF, prsEdgeOff);

  TIMER_Init_TypeDef timerInit = TIMER_INIT_DEFAULT;
  timerInit.enable = false;
  TIMER_Init(TIMER0, &timerInit);
  TIMER_TopBufSet(TIMER0, CMU_ClockFreqGet(cmuClock_TIMER0)/SAMPLING_FREQ);
}
```

Llistat 20.6: Configuració del DMA per obtenir dades de l'ADC

```
static void DMAConfig(void) {
  ...
  /* configure DMA for ADC reads */
  dma_cb_adc.cbFunc = dmaTransferCompleteADC;
  dma_cb_adc.userPtr = NULL;

  chnlCfg.highPri = false;
  chnlCfg.enableInt = true;
  chnlCfg.select = DMAREQ_ADC0_SINGLE;
  chnlCfg.cb = &dma_cb_adc;
  DMA_CfgChannel(DMA_CHANNEL_ADC, &chnlCfg);

  descrCfg.srcInc = dmaDataIncNone;
  descrCfg.dstInc = dmaDataInc2;
  descrCfg.size = dmaDataSize2;
  descrCfg.arbRate = dmaArbitrate1;
  descrCfg.hprot = 0;
  DMA_CfgDescr(DMA_CHANNEL_ADC, true, &descrCfg);
  ...
}
```

Llistat 20.7: Configuració del DMA per obtenir dades de l'ADC

```
void GPIO_EVEN_IRQHandler(void) {
  uint32_t aux;

  /* clear flags */
  aux = GPIO_IntGet();
  GPIO_IntClear(aux);

  LedOn();

  /* Activate DMA transfer */
  DMA_ActivateBasic(DMA_CHANNEL_ADC, true, false, (void*)DMAbufferADC, (void*)
      &(ADC0->SINGLEDATA), SAMPLES-1);

  /* Activate TIMER 0 */
  TIMER_CounterSet(TIMER0, 0);
  TIMER_Enable(TIMER0, true);
}
```

Llistat 20.8: Configuració del DAC perquè funcioni amb el PRS

```
static void DACConfig(void) {
  ...
  /* Use PRS channel 3 */
  initChannel.prsEnable = true;
  initChannel.prsSel = dacPRSSELCh3;
  DAC_InitChannel(DAC0, &initChannel, 1);
  ...
}
```

Llistat 20.9: Configuració del PRS i el Timer

```
static void DACConfig(void) {
  ...
  /* Configure PRS channel 3 trigger to be TIMER1*/
  PRS_SourceSignalSet(3, PRS_CH_CTRL_SOURCESEL_TIMER1,
      PRS_CH_CTRL_SIGSEL_TIMER1OF, prsEdgeOff);

  TIMER_Init_TypeDef timerInit = TIMER_INIT_DEFAULT;
  timerInit.enable = false;
  TIMER_Init(TIMER1, &timerInit);
  TIMER_TopBufSet(TIMER1, CMU_ClockFreqGet(cmuClock_TIMER1)/SAMPLING_FREQ);
}
```

Llistat 20.10: Configuració del PRS i el Timer

```
static void DMAConfig(void) {
  ...
  chnlCfg.highPri = false;
  chnlCfg.enableInt = true;
  chnlCfg.select = DMAREQ_DAC0_CH1;
  chnlCfg.cb = &dma_cb_dac;
  DMA_CfgChannel(DMA_CHANNEL_DAC, &chnlCfg);

  descrCfg.srcInc = dmaDataInc2;
  descrCfg.dstInc = dmaDataIncNone;
  descrCfg.size = dmaDataSize2;
  descrCfg.arbRate = dmaArbitrate1;
  descrCfg.hprot = 0;
  DMA_CfgDescr(DMA_CHANNEL_DAC, true, &descrCfg);
}
```

A continuació es configura el canal número 3 del PRS perquè obtingui el senyal del Timer 1 (Llistat 20.9). I per últim es configura el DMA perquè el seu *trigger* sigui el canal 1 del DAC (Llistat 20.10).

La ISR del botó 1, haurà d'engegar el DMA i el Timer 1 per engegar el procés de generar el senyal prèviament mostrejat (Llistat 20.11).

Llistat 20.11: ISR del botó 1

```
void GPIO_ODD_IRQHandler(void) {
  uint32_t aux;

  /* clear flags */
  aux = GPIO_IntGet();
  GPIO_IntClear(aux);

  /* Activate DMA transfer */
  DMA_ActivateBasic(DMA_CHANNEL_DAC, true, false, (void*) &(DAC0->CH1DATA), (
      void*)DMAbufferADC, SAMPLES-1);

  /* Activate TIMER 1 */
  TIMER_CounterSet(TIMER1, 0);
  TIMER_Enable(TIMER1, true);
}
```

21. Una aplicació completa

Ja va sent hora de fer una aplicació completa (senzilla) per il·lustrar tot el que hem anat aprenent durant el curs.

Anem a veure una aplicació sencera on ajuntarem unes quantes coses de les que hem vist fins ara. Farem una aplicació que segons la proximitat de la ma al sensor (o del sensor a una taula o un obstacle), faci pampallugues més ràpid o més lent el LED de la PCB. Farem servir el sensor de proximitat APDS-9960 connectat a la nostra placa de prototipat com ja vàrem fer a l'exemple d'I2C. Per tant, caldrà anar llegint cíclicament el sensor de proximitat i canviar el valor de PWM del LED segons el valor llegit.

Com que aquest projecte fa servir diferents perifèrics del microcontrolador i les seves llibreries, cal organitzar el codi perquè tot sigui entenedor i amb un manteniment senzill. Així, tindrem un conjunt de mòduls (se li diu mòdul a una parella de fitxers .c i .h) per manegar les diferents funcionalitats, sensors i perifèrics. Per últim, tindrem el fitxer *main.c* on hi ha la funcionalitat principal de l'aplicació.

21.1 Biblioteques

Aquesta aplicació fa servir quatre biblioteques per organitzar el codi, anem a descriure-les amb detall.

21.1.1 BSP

Habitualment en aquest mòdul s'hi posen les configuracions i inicialitzacions de diferents perifèrics o dispositius comuns a tot el sistema.

En aquest cas es gestiona algun dels rellotges del sistema, el pin corresponent al LED i es configura la sortida de la consola pel **printf**.

D'aquesta biblioteca es fa servir la funció **setupSWOForPrint()** i la funció **BSP_Init()**. Aquesta segona funció tant sols crida la primera i inicialitza el LED de la placa.

Llistat 21.1: Part de la funció **I2C_WriteRegister**

```
bool I2C_WriteRegister(uint8_t addr, uint8_t reg, uint8_t data) {
  ...
  I2C_Status = I2C_TransferInit(I2C0, &seq);

  while (I2C_Status == i2cTransferInProgress) {
    I2C_Status = I2C_Transfer(I2C0);
  }
  ...
}
```

Llistat 21.2: Funció PWM_Set()

```
void PWM_Set(uint8_t percentage) {
  uint32_t pwm_value;
  ...
  /* convert to percentage (0 to 100) to range 0 - PWM_FREQ */
  pwm_value = percentage * PWM_FREQ / 100;

  TIMER_CompareBufSet(TIMER1, 1, pwm_value);
}
```

21.1.2 I2C_Wrapper

Hem escrit una biblioteca senzilla per accedir al bus I2C fent servir la llibreria emlib. D'aquesta manera encapsulem tota la funcionalitat en dues funcions senzilles de fer servir per llegir o escriure un registre d'un dispositiu I2C (**I2C_WriteRegister** i **I2C_ReadRegister**) com ja es va fer a **Secció 15.1 - Exemple d'I2C**.

21.1.3 PWM

És una biblioteca molt senzilla per controlar el canal PWM que correspon al LED de la nostra placa. Només suporta aquest canal i només proporciona una funció per controlar el PWM, amb un paràmetre amb el percentatge de duty cycle del PWM (100% equival al LED sempre encès i 0% correspon al LED apagat). La funció **PWM_Init()** configura el Timer1 perquè funcioni com a generador de PWM, de manera molt similar a l'exemple **Secció 10.1 - Generar PWM**.

L'altra funció de la biblioteca és **PWM_Set()**, que rep el percentatge del duty cycle que ha d'estar a 1 i configura el Timer d'acord a aquest percentatge. Com que el Timer està configurat perquè compti fins a **PWM_FREQ** (que val 4096), cal passar del rang 0 – 100 a 0 – 4096. Un cop calculat el valor correcte, s'usa com a valor al comparador del *timer*.

21.1.4 APDS-9960

Aquesta biblioteca controla el sensor APDS-9960 via el bus I2C. Només gestiona la funcionalitat de sensor de proximitat (que pot fer força més coses).

La funció **APDS_9960_InitProximity()** (Llistat 21.3) inicialitza el sensor per a què funcioni com a sensor de proximitat. Cal consultar el Datasheet per veure els registres que s'escriuen i quins valors es posen [6].

La funció **APDS_9960_ReadProximity()** (Llistat 21.4) llegeix el registre d'estatus i si hi ha una

Llistat 21.3: Funció **APDS_9960_InitProximity()**

```
void APDS_9960_InitProximity() {
  // Enable Proximity detection
  // ENABLE <- 5 & 2 & 0 bits
  I2C_WriteRegister(DEVICE_ADDR, APDS_ENABLE_REG, 0x25);

  /* LED Strength to 100mA, Proximity Gain control to 8x */
  I2C_WriteRegister(DEVICE_ADDR, APDS_CTRL_1_REG, 0x0C);

  /* LED_BOOST 300% 0111_0001*/
  I2C_WriteRegister(DEVICE_ADDR, APDS_CTRL_2_REG, 0x71);
}
```

Llistat 21.4: Funció **APDS_9960_ReadProximity()**

```
bool APDS_9960_ReadProximity(uint8_t *p_data) {
  uint8_t status;
  bool ret = false;

  I2C_ReadRegister(DEVICE_ADDR, APDS_STATUS_REG, &status);

  if ((status & 0x02) != 0x00) {
    I2C_ReadRegister(DEVICE_ADDR, APDS_PDATA_REG, p_data);
    ret = true;
  }

  return ret;
}
```

dada disponible la llegeix, la retorna a través del paràmetre per referència i el retorn de la funció valdrà **True**; si no hi ha una dada disponible retorna **False**.

La biblioteca es basa en *polling* per saber si la dada a llegir és bona, comprovant el registre **status** i retornant fals si no s'ha pogut llegir. Es podria millorar la biblioteca fent que el sensor llenci una interrupció cada cop que tingui una dada nova (es veurà més endavant a **Secció 21.3 - Afegint-hi interrupcions**).

21.2 Funció principal

La funció principal de l'aplicació està en el propi **main()** i consta bàsicament d'un bucle sense fi on es va llegint el valor de proximitat del sensor (si està disponible) i s'ajusta el duty cycle del PWM segons aquest valor, de manera que el LED pampalluguegi més sovint quan més a prop estigui l'obstacle.

Com que hem escrit la funció **APDS_9960_ReadProximity()** de manera que retorni true o false segons si s'ha pogut llegir o no una dada, el codi ens queda molt senzill i llegible. Si s'ha pogut llegir, es treu per la consola de *debug* la dada i es converteix al rang adequat el valor llegit.

R Fixem-nos que la conversió de rang es fa forçant primer la multiplicació per 100 abans de la divisió per 256 per tal de no perdre precisió en la conversió. Com que treballem amb nombres enters, la divisió és entera i si ho féssim a l'inrevés obtindríem sempre valors 0, ja que <valor de 8 bits>/256 sempre dona 0 en una divisió entera i després encara que ho multipliquéssim per 100 seguiria donant 0..

Llistat 21.5: Funció principal

```
while(pdTrue) {
  ret = APDS_9960_ReadProximity(&p_data);

  if (ret == true) {
    printf("Proximity: %d\r\n", p_data);

    /* Convert from range 0 - 256 to 0 - 100 */
    PWM_Set((p_data * 100) / 256);
  }
}
```

21.3 Afegint-hi interrupcions

En els exemples que hem vist fins ara, tant de perifèrics integrats com de dispositius externs la comunicació es feia via *polling*, això és, preguntant tota l'estona si el dispositiu o perifèric té alguna dada disponible. L'altre manera de comunicar-se és mitjançant interrupcions, fent que el dispositiu o perifèric llenci una IRQ cada cop que té una dada disponible.

Per això, ens calen almenys, fer els següents canvis al nostre sistema:

- Connectar el pin d'IRQ del sensor a la nostra placa de desenvolupament
- Configurar el sensor **APDS-9960**, de manera que generi un senyal d'IRQ cada cop que tingui una nova dada disponible
- Habilitar les interrupcions al nostre microcontrolador perquè cridi la ISR corresponent.

21.3.1 Connexió del pin INT

Per aconseguir connectar el pin INT del sensor **APDS-9960** només cal fer servir un cinquè cable Dupont entre la placa de prototipat del sensor i la nostra placa de desenvolupament. Si mirem el manual de la placa de Silicon Labs, veurem que el pin PB12 es pot fer servir com a un GPIO normal i que generi una interrupció cada cop que tinguem un flanc.

Segons el *datasheet* [6, pàgina 3] del APDS-9960 el pin d'interrupció és de tipus *Open drain*. Això vol dir que aquest pin només pot forçar un valor '0' i li cal que el valor '1' estigui forçat per un pull-up. En el cas de la placa que tenim, ja incorpora aquesta resistència de pull-up, per tant, el pin de GPIO del microcontrolador no cal configurar-lo com entrada amb pull-up.

Si la PCB no tingues la resistència de pull-up, caldria configurar el pin com a pull-up ja que si no ens trobaríem que sempre estaria a '0', ja que cap dels dos dispositius el posaria a '1'.

21.3.2 Configurar el dispositiu APDS9960

Ara cal que el sensor generi un senyal d'IRQ cada cop que tingui una nova dada disponible. Veiem al *datasheet* [6, pàgina 11 i 20] que cal activar les interrupcions per proximitat (bit **PIEN** del registre **Status**) i si configurem el valor del camp **PPERS** del registre **Persistence** al valor 0 generarà una interrupció cada cop que hi hagi una dada nova.

Amb aquesta configuració segons el *datasheet* el sensor posarà una interrupció cada cop que hi hagi una dada nova disponible de proximitat.

21.3.3 Habilitar la interrupció corresponent

L'últim pas és habilitar la interrupció corresponent al pin d'entrada que hem triat. És una operació molt similar al que ja s'ha vist a l'apartat dels GPIOs (**Capítol 6 - GPIO**).

Un cop tenim tot preparat, cal també modificar la nostra funció principal perquè no faci *polling*, que és el que volem evitar. Com que ara tenim una *ISR* que s'executarà cada cop que que hi hagi una dada nova al sensor, tenim vàries opcions, en destaquem dues:

- Fer la lectura del registre del sensor des de la pròpia ISR
- Activar un *flag* que ens indiqui que podem fer la lectura des de la funció principal.

```
GPIO_PinModeSet(gpioPortB, 12, gpioModeInput, 1); /* IRQ from APDS_9960 */
```

Llistat 21.6: Nova funció d'initialització del APDS_9960

```
void APDS_9960_InitProximity_IRQ() {
  //Enable Proximity detection
  // ENABLE <- 5 & 2 & 0 bits
  I2C_WriteRegister(DEVICE_ADDR, APDS_ENABLE_REG, 0x25);

  /* LED Strength to 100mA, Proximity Gain control to 8x */
  I2C_WriteRegister(DEVICE_ADDR, APDS_CTRL_1_REG, 0x0C);

  /* LED_BOOST 300% 0111_0001*/
  I2C_WriteRegister(DEVICE_ADDR, APDS_CTRL_2_REG, 0x71);

  /* Generate IRQ every data valid */
  I2C_WriteRegister(DEVICE_ADDR, APDS_PERSISTENCE_REG, 0);
}
```

Llistat 21.7: Habilitar l'interrupció del pin corresponent

```
GPIO_IntConfig(gpioPortD, 8, false, true, true);
NVIC_EnableIRQ(GPIO_EVEN_IRQn);
```

Com ja s'ha comenta anteriorment (**Capítol 7 - Controlador d'interrupcions**) posar gaire codi dins una ISR no és aconsellable, així que per aquest exemple farem servir la segona opció proposada. Per tant, afegirem una variable de tipus *bool* al projecte que farem servir com a *flag*. Aquest *flag* el posarem a **true** dins la ISR i la funció principal l'esborrarà cada cop que el faci servir (Llistats 21.8 i 21.9).

El que veiem a la funció **main()** és que el microcontrolador estarà la major part del temps esperant a que el *flag* es posi a *true* per tal de continuar i poder fer l'operació de lectura del sensor i canviar el duty cycle del PWM del LED. Tota aquesta estona que el microcontrolador està esperant-se sense poder fer res, es podrà aprofitar per posar-lo en algun mode de baix consum tal com veurem al **Capítol 35 - Baix cosum**.

Llistat 21.8: ISR amb el *flag*

```
void GPIO_EVEN_IRQHandler(void) {
  /* clear flags */
  aux = GPIO_IntGet();
  GPIO_IntClear(aux);

  signal = true;
}
```

Llistat 21.9: Funció principal amb suport d'interrupcions

```
main() {
  ...
  while(signal == false);
  signal = false;

  ret = APDS_9960_ReadProximity(&p_data);
  ...
}
```

Aquesta pàgina està en blanc expressament, tot va bé.

IV FreeRTOS

22 Conceptes bàsics de FreeRTOS 127
22.1 Temps Real
22.2 Tasques
22.3 El temps en un RTOS
22.4 Interrupcions a FreeRTOS

23 Primer exemple amb FreeRTOS 137

24 Controlant el temps a les tasques . . . 139
24.1 Un exemple amb vTaskDelayUntil()

25 Comunicació entre tasques 143
25.1 Semàfors
25.2 Cues
25.3 Mutex
25.4 *Event Groups*
25.5 Conjunt de cues *Queue Sets*
25.6 Notificacions a tasques
25.7 Comparant temps de resposta

26 Exemple amb la UART i interrupcions 159

27 Una aplicació completa amb FreeRTOS 163
27.1 Tasques
27.2 Modificant el *wrapper* d'I2C
27.3 Analitzant les diferències

28 Ús del watchdog en RTOS 167

29 *Drivers* en multi-tasca 169

Aquesta pàgina està en blanc expressament, tot va bé.

22. Conceptes bàsics de FreeRTOS

En el Firmware per sistemes encastats que hem vist fins ara es basen en un bucle infinit on es van executant les tasques a fer. Això acostuma a ser prou bo per sistemes senzills, com ara llegir d'un ADC i decidir alguna cosa, o actuar sobre una sortida segons el valor d'un sensor, etc.

Per sistemes més complexos o amb requeriments crítics, s'acostuma a fer servir un Sistema Operatiu per gestionar diferents tasques.

> **R** Un sistema és de Temps Real quan el temps de resposta del sistema a un event extern està fitada. És a dir, es pot saber i està garantit el temps total entre que succeeix un esdeveniment i el sistema genera una sortida.
>
> Un exemple típic és el d'un airbag. El temps màxim, sigui el que sigui que el sistema estigui fent entre que es detecta l'accident i es dispara l'airbag està garantit i limitat.
>
> Un Sistema Operatiu de Temps Real (RTOS en anglès) és un Sistema Operatiu que està dissenyat per garantir aquesta fita de temps.

En aquest llibre farem servir FreeRTOS [51], un RTOS[1] de codi obert àmpliament utilitzat. En el cas de EFM32 i STM32 els fabricants ens proporcionen el *porting* de FreeRTOS a les seves plataformes.

> **R** Se'n diu *porting* al fet d'adaptar un codi a una plataforma específica. Per exemple, en el cas del FreeRTOS, cal adaptar una sèrie de funcions per a que tot el sistema funcioni, com ara la gestió dels rellotges, la gestió de tasques, etc.

En tot OS[2] (i RTOS) les feines a fer per part del sistema es reparteixen en diferents tasques (tasks en anglès). Aquestes tasques s'executen "simultàniament" i, per tant, cal repensar bé tot el sistema

[1] *Real-Time Operating System*, Sistema Operatiu de Temps Real
[2] *Operating System*, Sistema Operatiu

abans de començar a escriure el codi.

En sistemes encastats, un SO (o RTOS) no ofereix totes les funcionalitats a les que estem acostumats quan sentim parlar d'un SO. Així, normalment el que ens ofereix un RTOS és:

- Gestió de tasques: Creació, execució, estat de les tasques, prioritats de tasques, etc.
- Comunicació entre tasques: semàfors, cues, etc.
- Gestió de temps: Timers, *timeouts*, *delays*, etc.

Les tasques són les unitats bàsiques de funcionament i és on s'implementen les funcionalitats del sistema.

22.1 Temps Real

Com ja s'ha dit, FreeRTOS és un Sistema Operatiu de Temps Real, que vol dir que totes les seves operacions tenen un temps d'execució fitat de manera que permet construir aplicacions de Temps Real.Aquest temps fita ve donat perquè tot les funcions del Sistema Operatiu son deterministes, això és, es pot saber *a priori* quin temps tardaran a executar-se i, en principi, no han de dependre de factors externs. Així, per exemple, desbloquejar una tasca que està depenent d'un semàfor sempre tardarà el mateix temps per una plataforma donada, fer un *context switch* entre tasques el mateix, encara que hi hagi dues o cinquanta tasques preparades, etc.

Això és especialment important quant la nostra aplicació ha de reaccionar ràpidament a algun esdeveniment extern, ja que podrem mesurar i/o calcular la fita màxima de la operació crítica i podrem confiar en que el sistema sempre estarà fitat per aquest valor siguin les condicions que siguin.

Quan es parla de temps real, usualment se'n fa una classificació en dos tipus: *soft real-time* i *hard real-time*. S'entén per *soft real-time* aquella aplicació en que existeix alguna restricció de temps en algun moment, però aquesta restricció es pot incomplir sense que l'aplicació falli. Un exemple podria ser una aplicació on cal mostrar vídeo per una pantalla; aquesta aplicació tindrà una fita tal que permeti mostra rel vídeo de forma prou suau, però si mai es per un *frame* l'aplicació no falla, si no que l'usuari veurà un artefacte estany a la pantalla. En canvi, un sistema que es defineixi com *hard real-time* no pot incomplir en cap moment aquesta restricció temporal. Un exemple típic és l'*airbag* d'un cotxe, que cal que es dispari en el moment adequat i que, en canvi, pot provocar danys si es dispara més tard.

Cal fer notar que precisament per la condició de cricitat i de determinisme, hem d'escriure les funcions d'ISR el més curt possibles ja que, a la majoria de les plataformes, quan s'està executant una ISR estan desactivades les IRQs i el RTOS està "venut", en el sentit que no pot interrompre ni controlar el temps que es gasta en la ISR.

22.2 Tasques

Una tasca és el que es coneix com a procés en els Sistemes Operatius de propòsit general. Per tant, una tasca serà l'estructura mínima de codi en que es divideix una aplicació. Aquestes tasques seran les que el planificador o *scheduler* del Sistema Operatiu anirà executant i retirant d'execució segons certes condicions que veurem a continuació.

Bàsicament una tasca pot estar en l'estat *Running* (executant-se), *Ready* (disponible per ser executada) o *Blocked* (no preparada perquè està esperant a algun esdeveniment)[3]. A la Figura 22.1 es pot veure quins estats pot tenir una tasca en FreeRTOS [52, pàgina 92].

[3]També hi ha l'estat *Suspended* on la pròpia tasca demana sortir de la llista de *Ready*

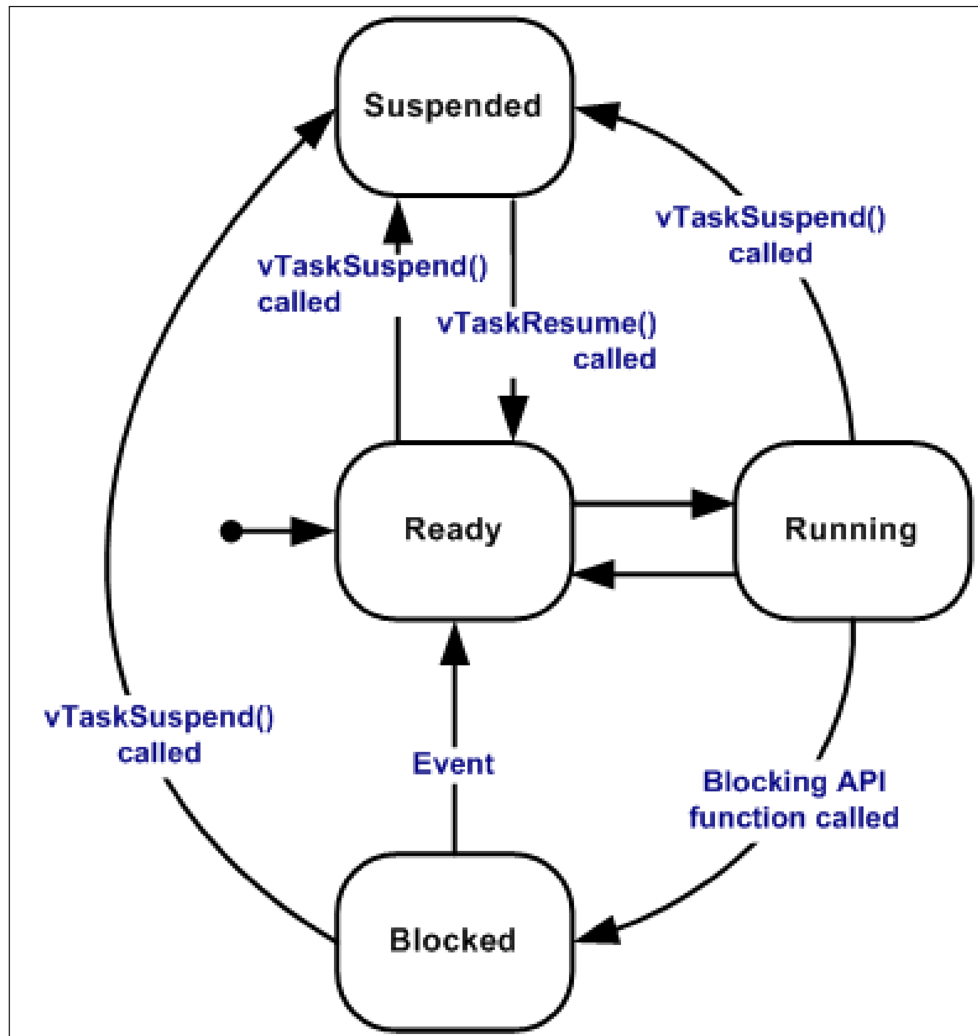


Figura 22.1: Estats possibles d'una tasca
Estats possibles d'una tasca. © FreeRTOS

Llistat 22.1: Esquelet d'una tasca

```
static void OneTask(void *pParameter) {
  (void) pParameter;
  /* Init */

  while(1) {
    /* bucle infinit */
  }
}
```

Les tasques són funcions amb un bucle infinit i una inicialització prèvia i en cap cas la funció pot retornar. Si una tasca ja no és necessària, es pot eliminar per part el RTOS, però no pot acabar retornant la funció per si mateixa. Pel cas de FreeRTOS, tenen l'aspecte que es mostra al Llistat 22.1.

Una aplicació basada en un RTOS és un conjunt de tasques funcionant concurrentment i comunicades entre elles. El RTOS ens dona diferents funcions que ens permeten: comunicar tasques entre elles o bloquejar-se per un cert temps.

22.2.1 Prioritats

Quan es creen les tasques, a cada una se li assigna una certa prioritat. Aquesta prioritat marcarà quina de les tasques de l'estat *Ready* passarà a executar-se *Running* a cada moment. A FreeRTOS les tasques amb una prioritat d'un valor més baix tenen menor prioritat (0 és el valor més baix i menys prioritari). Diferents tasques poden tenir la mateixa prioritat.

Existeix una tasca *Idle* que s'executa cada cop que cap altra tasca està a l'estat *Ready*. Aquesta tasca té la prioritat més baixa, de valor 0 i per tant només s'executa quan cap tasca està llesta. La tasca *Idle* sempre està disponible per executar-se a qualsevol moment.

Per saber més sobre prioritats i com assignar-les a les tasques, veieu **Capítol 49 - Assignació de prioritats**.

22.2.2 L'ús de l'*stack* en un S.O.

Per crear una tasca, un dels paràmetres que es passa a la funció **xTaskCreate** és la mida de l'*stack* per la tasca que s'està creant.

Se'n diu *stack* a la regió de memòria assignada a una tasca que s'utilitza bàsicament per dues coses:

- guardar els valors dels paràmetres de les diferents crides a funcions que faci la tasca
- emmagatzemar el seu context quan el SO retira la tasca d'execució.

Aquest *stack* normalment es gestiona com una pila (amb els mètodes *push* i *pop*) i acostuma a situar-se al final de la memòria i "créixer cap avall".

A priori, és molt difícil saber quant *stack* farà servir una tasca, així que el més habitual és donar un valor per defecte prou gran i després, durant el funcionament normal, observar quanta se'n fa servir realment.

Per saber el nivell màxim a on s'ha arribat a l'*stack* de cada tasca, FreeRTOS fa servir un mètode una pel peculiar: a l'inicialitzar la tasca (i el seu *stack*) omple l'*stack* amb un valor predeterminat i conegut. Després, quan ja està funcionant el nostre sistema, només cal comptar quantes posicions

de l'*stack* mantenen el valor conegut. Si s'acosta a 0 voldrà dir que la tasca està fent servir la majoria de l'espai de l'*stack*.

Per activar aquesta funcionalitat cal editar el fitxer **FreeRTOSConfig.h** i definir la macro **configCHECK_FOR_STACK_OVERFLOW** amb el valor 2 i la macro **uxTaskGetStackHighWaterMark** a 1.

Activant la primera de les macros, el sistema operatiu també controla que no s'intenti accedir fora de l'*stack*. D'aquesta manera podem saber si cal més *stack*, però no si ens en sobra ni quanta. En activar aquesta opció cal definir una funció de hook que s'anomeni **vApplicationStackOverflowHook()** i que es cridarà cada cop que es detecti un error en accedir l'*stack*.

22.3 El temps en un RTOS

Habitualment un RTOS ha de portar un control del temps per decidir quina tasca s'ha d'executar, saber quan desbloquejar una tasca, etc.

Normalment es basen en el concepte de *tick*. Un *tick* es genera de forma periòdica (normalment fent servir algun Timer) i la freqüència d'aquest *tick* ens marca el temps mínim que pot manegar el RTOS.

En el cas de FreeRTOS, a cada *tick* es deixa d'executar la tasca que està *running* i s'executa el planificador del RTOS. Si aquest planificador detecta que una tasca més prioritària està a punt per ser executada (estat *ready*), la posarà a executar (*running*) enlloc de la que hi havia fins feia un moment.

La freqüència d'aquest *tick* pot variar molt segons el sistema que tinguem, però acostuma a anar entre els 1000 Hz i els 50Hz. Segons la freqüència del *tick* tindrem un sistema més ràpid de resposta en segons quins casos a canvi d'alentir lleugerament l'execució general del sistema i de tenir menys resolució en les funcions de control de temps.

22.3.1 Funcions per controlar el temps

Tot OS proporciona una sèrie de funcions per gestionar el temps, això és:

- bloquejar una tasca durant un cert temps determinat.
- controlar el *timetout* a certes crides del mateix.

Delays

Les funcions **vTaskDelay()** i **vTaskDelayUntil()** són les dues funcions que ens permeten bloquejar la tasca que la crida durant el temps que s'especifica com a paràmetre.

Aquestes dues funcions faran que la tasca entri immediatament a l'estat *blocked* i restarà en aquest estat durant tota l'estona que s'especifiqui. Un cop hagi passat aquest temps la tasca passarà a l'estat *Ready*.

El paràmetre que rep la funció **vTaskDelay()** és el nombre de *ticks* que es vol que la tasca estigui *blocked*. Com més precís sigui el *tick* del sistema (més freqüència) més acurat podrà ser aquest *delay*.

Cal veure que segons la prioritat de la tasca en qüestió i de la prioritat de la tasca que s'estigui executant, aquesta tasca pot romandre en l'estat *Ready* un temps indefinit.

Timeout

A les funcions d'accedir a un recurs compartit, com un semàfor o una cua, apareix un paràmetre que és el temps (en *Ticks*) que la tasca que accedix esperà a poder realitzar l'operació. Així, si

s'especifica un temps de 100 mil·lisegons per llegir d'una cua, la funció retornarà passat aquest temps si la cua està buida i ningú hi ha escrit res o retornarà tant bon punt algú hi escrigui alguna dada.

22.4 Interrupcions a FreeRTOS

FreeRTOS deixa el maneig de les interrupcions a mans del desenvolupador, demanant unes certes condicions.

Cal tenir en compte que les interrupcions són esdeveniments totalment asíncrons i imprevisibles i que prenen el control de forma automàtica. Això fa que mentre està funcionant una ISR el *kernel* del Sistema Operatiu no es pot executar i que, quan acabi d'executar la ISR, si no fem res, tornarà el control cap a la tasca que s'estava executant. Això pot provocar que una ISR alliberi un recurs o posi disponible una dada i que una tasca d'alta prioritat passi a l'estat *Ready* però el *kernel*, com que no s'executa, no pugui passar-li l'execució i se segueixi amb la taca menys prioritària que s'estava executant.

Per això les funcions per accedir a recursos com semàfors o cues des d'una ISR tenen un paràmetre extra, que informa si s'ha despertat una tasca més prioritària. Si és el cas, cal que el codi de la ISR faci un **portYIELD_FROM_ISR()** per cridar al *kernel* del Sistema Operatiu (veure Llistat 22.2.

A la Figura 22.2 hi ha un diagrama de seqüència d'un exemple amb dues tasques: la Tasca 1 és la més prioritària i es bloqueja esperant rebre una dada per una cua. Quan es bloqueja s'executa la Tasca 2. Mentre s'està executant arriba una IRQ que posa una dada a la cua de la Tasca1 i retorna. Com que el *kernel* no pot obté el control, es segueix executant la Tasca 2, menys prioritària. Al diagrama de la Figura 22.3 succeeix el mateix que abans, però ara la ISR crida a **portYIELD_FROM_ISR()** en acabar i llavor es passa a executar el *kernel* i aquest dona l'execució a la Tasca 1. Aquest és el funcionament correcte que s'espera del sistema.

Aquesta funció de *yield* retorna de la ISR i executa el *kernel* si la variable passada té un valor diferent a **pdFALSE**.

Sempre es diu que les ISRs han de ser el més curtes possibles, això és pels següents motius:

- Mentre s'està executant una ISR no es pot executar cap tasca, per molt prioritària que sigui.
- Depenent de l'arquitectura, mentre s'està executant una ISR la resta d'IRQs estan desactivades.
- Algunes arquitectures poden permetre anidar interrupcions, cosa que augmenta la complexitat i la incertesa de tot el sistema. Quan més curta sigui la ISR més improbable que això passi.

Llistat 22.2: Codi ISR d'exemple

```c
void any_IRQHandler(void) {
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  ...
  /* Toggle semaphore */
  xSemaphoreGiveFromISR(semaphore_button_0, &xHigherPriorityTaskWoken);

  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

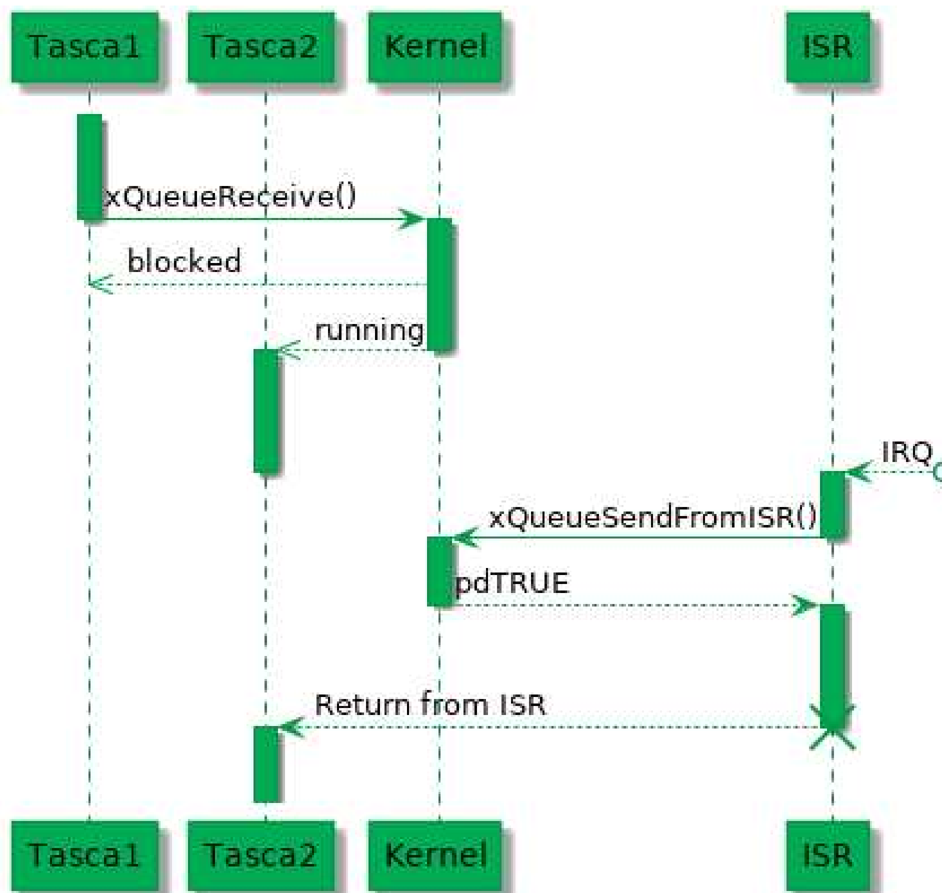


Figura 22.2: Diagrama de seqüència de dues tasques

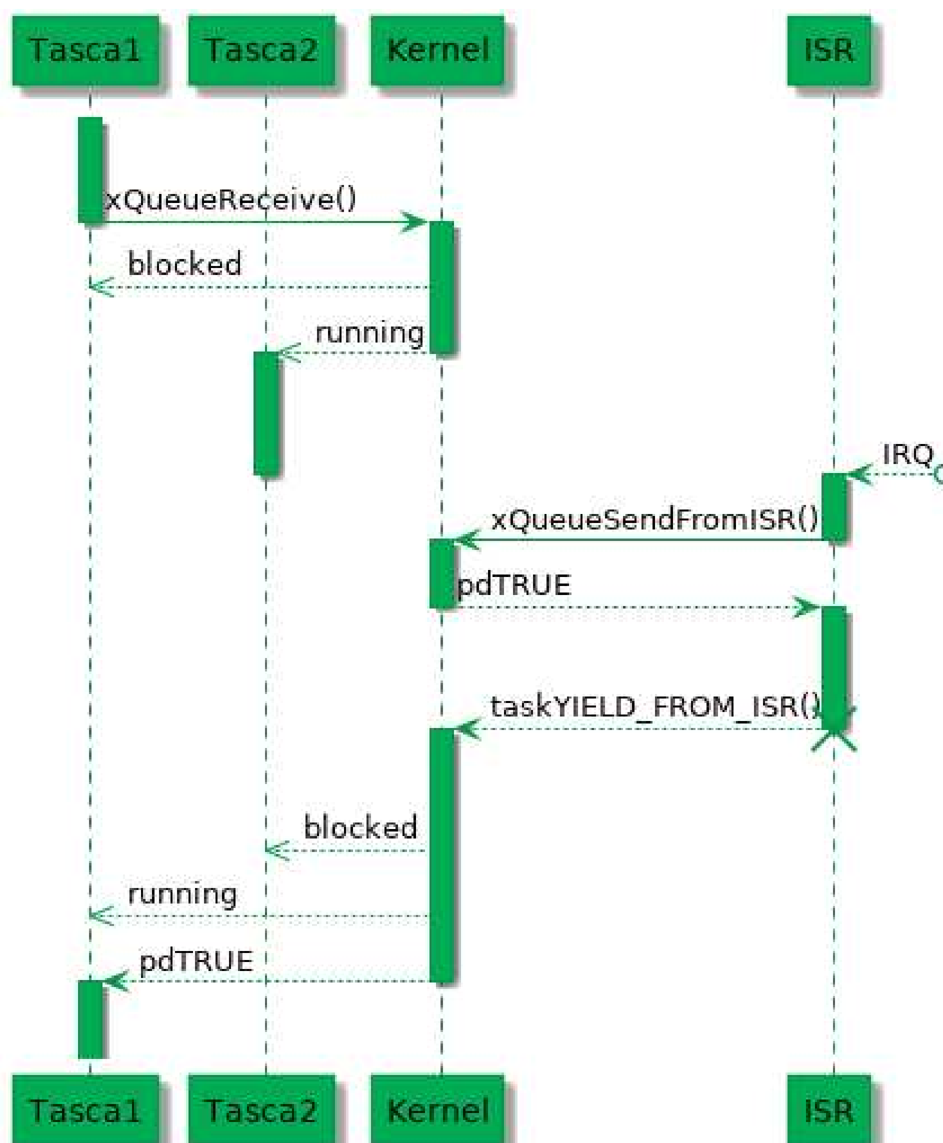


Figura 22.3: Diagrama de seqüència de dues tasques correcte

És per això que les bones pràctiques diuen que el codi dins una ISR hi hagi poc codi i es facin servir semàfors o cues per notificar tasques on s'executin les operacions pertinents amb les prioritats adequades.

Aquesta pàgina està en blanc expressament, tot va bé.

23. Primer exemple amb FreeRTOS

El primer exemple és el nostre vell conegut 'Hello World' per embedded, és a dir, blinkar un LED.

L'exemple consisteix en una sola tasca que s'encarrega de blinkar el LED. Com totes les tasques, consisteix en un bucle infinit on s'inclou tota la funcionalitat de la tasca (Llistat 23.1).

Aquesta tasca fa servir una funció del RTOS (**vTaskDelay()**) que bloqueja la tasca per un determinat temps. Així doncs, aquesta tasca tant sols farà Toggle al LED cada mig segon. Cal fixar-se que aquesta funció rep com a paràmetre el nombre de *ticks* a esperar-se. Aquest número es calcula amb la macro **pdMS_TO_TICKS()**, que passa de mil·lisegons a *ticks*.

Al **main()** el que veiem és que, després de la inicialització habitual hem de crear les tasques que tingui el nostre sistema i tot seguit engegar el FreeRTOS (Llistat 23.2).

La funció **xTaskCreate()** rep diferents paràmetres:

1. Punter a la funció que implementa la tasca
2. Nom que li posem a la tasca (per debug)
3. *stack* reservat per la tasca (veure **Subsecció 22.2.2 - L'ús de l'*stack* en un S.O**)
4. Punter a paràmetres per la tasca

Llistat 23.1: Tasca TaskLedToggle per FreeRTOS

```
static void TaskLedToggle(void *pParameter) {
  (void) pParameter;

  for (;;) {
    LedToggle();
    vTaskDelay(pdMS_TO_TICKS(500));
  }
}
```

Llistat 23.2: Main HelloWorld per FreeRTOS

```
main() {
  ...
  /* Create our first task */
  xTaskCreate(TaskLedToggle, (const char *) "LedToggle",
        configMINIMAL_STACK_SIZE, NULL, TOGGLE_TASK_PRIORITY, NULL);

  /* Start FreeRTOS Scheduler */
  vTaskStartScheduler();
  ...
}
```

5. Prioritat de la tasca
6. *Handle* a la tasca creada

Aquesta funció retorna **pdPASS** si s'ha creat la tasca o un error en cas contrari.

La funció **vTaskStartScheduler()**, que en condicions normals no retorna mai, comença l'execució del FreeRTOS i aquest, al seu torn, executarà la nostra tasca.

24. Controlant el temps a les tasques

Habitualment les tasques dins d'un sistema amb un Sistema Operatiu s'executen periòdicament, de manera que la tasca fa la seva feina i després demana suspendre's per un temps determinat. Les tasques, doncs poden cridar a la funció **vTaskDelay()** per demanar al *kernel* que la suspengui un determinat temps passat com a paràmetre. Per exemple, al codi FreeRTOS_Blink la tasca principal te l'estructura del Llistat 24.1. En aquest cas senzill, la tasca anirà canviant el LED d'encès a apagat cada 500 mil·lisegons

Però què passaria si el temps d'execució de la tasca fos variable? Si tenim un codi més complicat que segons les condicions tardi més o menys temps, es tindrà que la tasca tarda un cert temps variable a executar-se i després es suspèn durant 500 mil·lisegons. Això farà que la seva periodicitat varii a cada execució de la tasca, cosa que pot ser inacceptable segons els casos.

Per això FreeRTOS proporciona la funció **vTaskDelayUntil()** que te en compte el tems utilitzat per la tasca abans de suspendre-la, de manera que es torni a cridar exactament en el període de temps desitjat.

Llistat 24.1: Tasca de l'exemple FreeRTOS_BlinkTask

```
static void TaskLedToggle(void *pParameter) {
  for (;;) {
    LedToggle();
    vTaskDelay(pdMS_TO_TICKS(500));
  }
}
```

24.1 Un exemple amb vTaskDelayUntil()

A l'exemple FreeRTOS_Delay es fa servir la funció **vTaskDelayUntil()** dins una tasca que cada cop que s'executa un cicle te un temps diferent d'execució. Això és fa amb una crida a **vTaskDelay()** amb un retard aleatori amb valors entre 0 i 300 mil·lisegons. Després d'aquesta execució es crida la funció **vTaskDelayUntil()** per suspendre de manera que la seva periodicitat sigui sempre de 500 mil·lisegons (veure Llistat 24.2).

Per comprova el correcte funcionament es treu per la consola de *debug* el *tick* en que s'està executant i el temps aleatori que es gasta per la tasca (veure Figura 24.1). Es pot veure com, a la primera columna, el tick on s'executa la tasca és sempre múltiple de 64 i que el temps que està executant-se la tasca va variant (segona columna).

24.1.1 Comprovació amb l'oscil·loscopi

Es pot aprofitar que l'exemple fa *toggle* del LED el GPIO també està connectat a un pin del connector d'expansió de la placa de desenvolupament (al pin 15 del *Expansion header*). A més, s'ha reduït el temps del període a 100 mil·lisegons i un *delay* aleatori d'entre 0 i 75 mil·lisegons. Així es pot comprovar amb l'oscil·loscopi que el senyal generat és prou bo i estable (veure Figura 24.2).

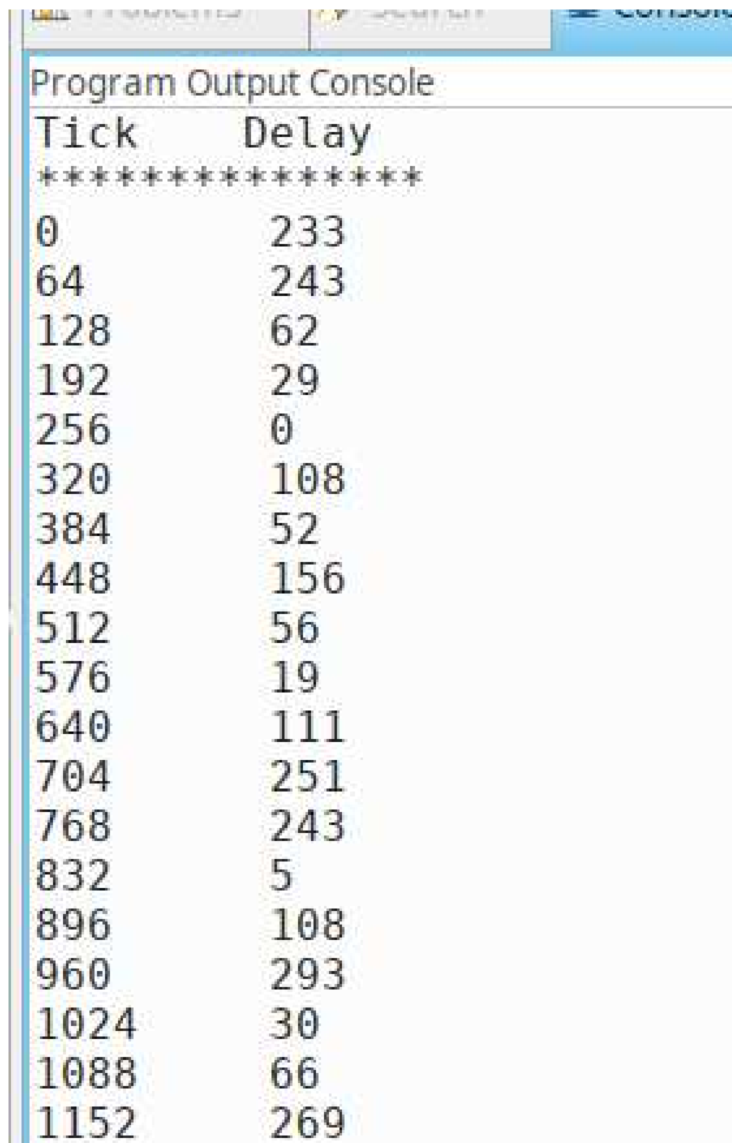


Figura 24.1: Consola amb la sortida de l'exemple FreeRTOS_Delay

Llistat 24.2: Tasca de l'exemple FreeRTOS_Delay

```
static void TaskLedToggle(void *pParameter) {
  ...
  for (;;) {
    LedToggle();
    printf("%lu", xTaskGetTickCount());

    /* Random delay from o to 300 ms. */
    random_time = rand() % 300;
    printf("\t %lu\r\n", random_time);
    delay_time = pdMS_TO_TICKS(random_time);
    vTaskDelay(delay_time);

    /* We want the task exactly every 500 milliseconds independently
    * from the (different) execution time */
    vTaskDelayUntil(&previous_tick_time, pdMS_TO_TICKS(500));
  }
}
```

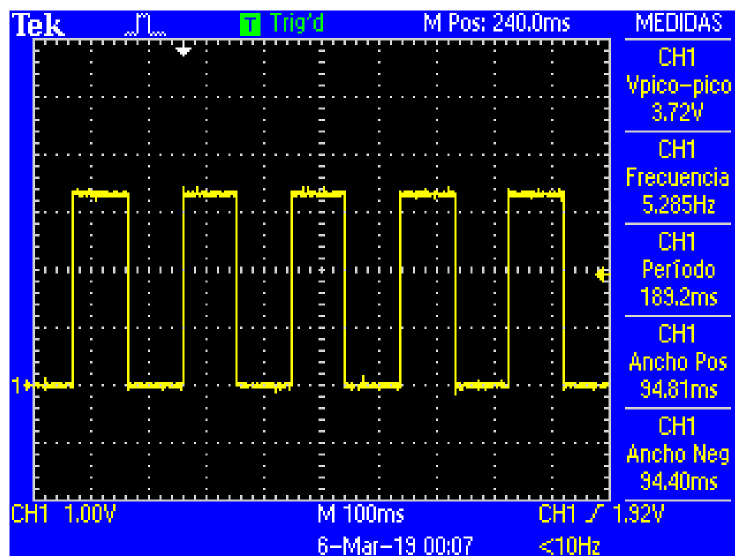


Figura 24.2: Captura de l'oscil·loscopi de l'exemple vTaskDelayUntil()

25. Comunicació entre tasques

En aquest capítol es detallen diferents mecanismes de sincronització i comunicació entre tasques en un Sistema Operatiu encastat.

Qualsevol RTOS que porti aquest nom ens oferirà una sèrie de mecanismes per a comunicar tasques entre elles. Els mecanismes més habituals són:

- Semàfors: Una tasca no pot agafar un semàfor fins que una altre no l'allibera.
- Cues: Permeten enviar informació d'una tasca a una altra.
- *Mutex*: Permet protegir un recurs compartit de manera que només una tasca el faci servir en un moment donat.

N'hi ha d'altres, com esperar i enviar esdeveniments o grups d'esdeveniments, *mailboxes*, senyals, etc. que acostumen a ser pròpies de cada OS concret.

25.1 Semàfors

Un semàfor és un dels mecanismes de comunicació entre tasques mes habituals dels que ofereix un OS. En essència el funcionament d'un semàfor és tal que una tasca prova d'agafar el semàfor i es quedarà esperant que una altra tasca doni el semàfor o ho tornarà a provar més endavant [53, pàgina 244] [54, pàgina 187].

Habitualment es fan servir per sincronitzar almenys dues tasques que comparteixen el semàfor o per protegir una secció crítica.

25.1.1 Semàfors a FreeRTOS

A FreeRTOS tenim diferents tipus de semàfors:

- *Binary*: pot tenir només l'estat 'agafat' o 'donat'. Es fan servir per sincronitzar tasques.
- *Counting*: s'emmagatzema un número, que s'incrementa en 'donar' el semàfor i es redueix en 'agafar' el semàfor. Sempre que tingui un valor positiu es podrà 'agafar' el semàfor. Serveix per portar un compte del nombre de recursos disponibles.

Llistat 25.1: Tasca amb semàfor d'exemple

```
static void TaskLedToggle(void *pParameter) {
  (void) pParameter;

  for (;;) {
    xSemaphoreTake(semaphore_button_0, portMAX_DELAY);
    LedToggle();
  }
}
```

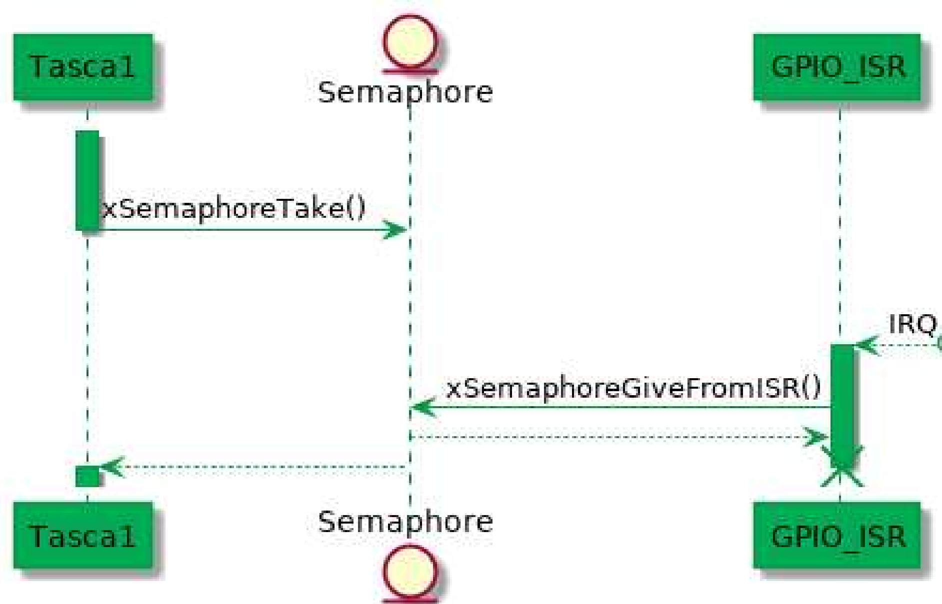


Figura 25.1: Diagrama de seqüència de l'exemple amb semàfors

.

- *Mutex*: són una variant dels semàfors binaris que inclouen mecanismes d'herència de prioritats. Es fan servir per implementar l'exclusió mútua. En parlarem més endavant, a **Secció 25.3 - Mutex**.

25.1.2 Exemple amb semàfors

A l'exemple hi ha una tasca que es queda esperant a agafar un semàfor i quan l'aconsegueix fa un *toggle* del LED (veure Llistat 25.1 i el diagrama de seqüència 25.1).

La funció **main()** crida la funció **BSP_Init()** que configura els GPIOs corresponents als botons i es registra una ISR (**GPIO_EVEN_IRQHandler**()) pel botó 0, que 'dóna' el semàfor en quan es prem el corresponent botó a la PCB (Llistat 25.2).

Després es crea el semàfor que compartiran la tasca i la ISR i tot seguit es crea la tasca com ja hem vist a l'exemple anterior.. Per últim s'engega el *kernel* del RTOS.

Cal notar que les funcions per agafar o donar un semàfor són diferents segons estiguem a una tasca o a una ISR. En el cas de la ISR, la funció de donar al semàfor ens indica si hi ha alguna tasca que cal desbloquejar perquè l'està esperant. En cas que sigui cert, cal fer un *yield* des de la ISR per a que el planificador (*scheduler*) del RTOS pugui actuar immediatament.

Llistat 25.2: ISR del botó 0

```
ISR() {
  ...
  /* Toggle semaphore */
  xSemaphoreGiveFromISR(semaphore_button_1, &xHigherPriorityTaskWoken);
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Tant en aquest exemple com en els següents el *timeout* de les crides és **portMAX_DELAY**. Aquesta macro serveix per indicar a la funció que es vol esperar un temps infinit a que l'operació es pugui realitzar. En aquest cas, la funció cridada no retornarà fins que pugui executar l'acció i bloquejant la tasca el temps necessari. D'aquesta mena de crides a funcions se'n diu crides bloquejants.

Exercici 25.1 Es pot provar d'implementar un codi que blinki el LED tantes vegades com vegades s'ha premut el botó 0. És pot fer amb un semàfor tipus *counting semaphore*. ■

25.2 Cues

Hem vist que els semàfors són útils per sincronitzar tasques i per protegir zones d'exclusió mútua, però no ens donen cap solució senzilla per enviar informació o dades d'una tasca a una altra. Aquesta comunicació és per les cues [52, pàgina 102].

Ens podem imaginar una cua com un recurs compartit entre dues o més tasques, on unes poden escriure-hi i d'altres hi poden llegir dades. Habitualment (en FreeRTOS és així), les cues s'implementen amb una estructura tipus FIFO (First-In First-Out) protegida de tal manera que no hi hagi cap *race condition* durant el seu funcionament[1].

A més, per tal de poder implementar sistemes d'una forma senzilla, els accessos a les cues poden ser bloquejants: la tasca que fa l'accés es quedarà bloquejada fins que pugui fer l'accés (esperar a poder escriure una dada, donat que no podia perquè la cua era plena) o esperar fins que hi hagi una dada (perquè s'ha intentat llegir de la cua quan aquesta era buida).

Les operacions habituals a una cua son:

- Crear una cua (*create*): normalment cal especificar quin tipus de dades ha d'emmagatzemar la cua i quants espais o llocs cal preparar.
- Inserir una dada (*send*): afegir (si hi ha lloc) una dada nova a la cua.
- Llegir dada (*receive*): treure una dada (si n'hi ha) de la cua.

Altres operacions poden ser:

- Mirar si hi ha una dada disponible (sense llegir-la) (*peek*).
- Esborrar tot el contingut de la cua (*reset*).
- Inserir una dada al principi de la cua (sense complir que la cua és una FIFO).

Les cues són recursos que poden ser compartits per vàries tasques, podent-hi escriure diferents tasques o ISRs i poder llegir-la també diferents tasques, tot i que això de tenir múltiples tasques llegint d'una cua no és gaire habitual.

Per saber quina mida han de tenir les cues, veieu **Capítol 50 - Mida de les cues**.

[1] S'anomena *race condition* al malfuncionament d'un codi donat que està executant-se en un entorn multitasca. Aquesta mena d'errors poden ser molt difícils de detectar i arreglar

Llistat 25.3: Part del codi d'una de les ISRs

```
...
/* Send the data to the Queue */
xQueueSendFromISR(queue_buttons, (void* ) &new_delay,
                 &xHigherPriorityTaskWoken);

/* Awake a task ? */
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
...
```

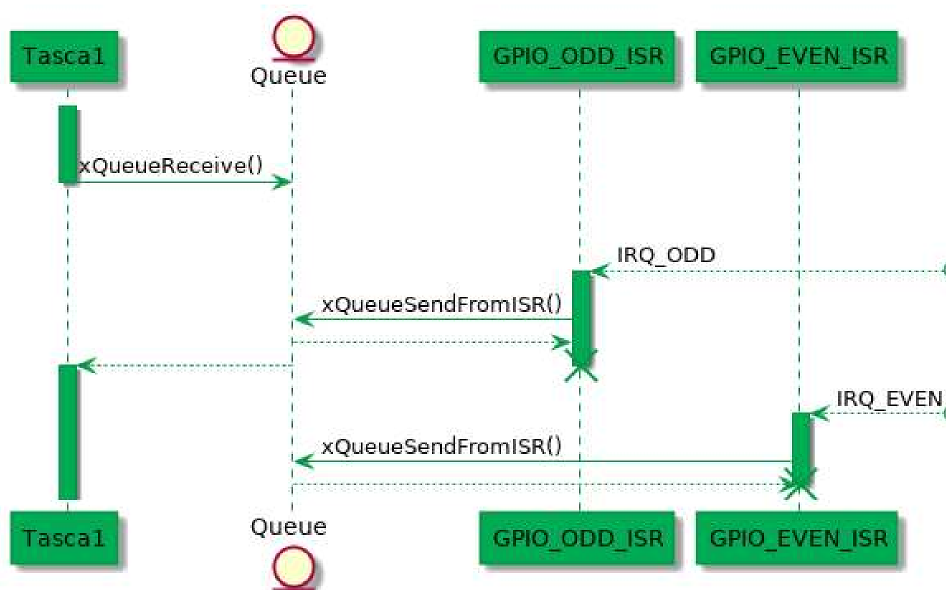


Figura 25.2: Diagrama de seqüència de l'exemple de cues

25.2.1 Exemple amb cues

A l'exemple de cues hi ha una sola tasca que fa *blinkar* el LED segons una variable. Aquesta variable s'obté de llegir (o intentar-ho) una cua. Aquesta cua (anomenada *queue_buttons*) l'escriuen les dues ISR associades als dos botons. Una envia el valor corresponent a 250 ms i l'altre ISR envia el valor que correspon a 1000 ms (Llistat 25.3 i diagrama de seqüència 25.2).

En aquest cas, la tasca fa servir la funció **xQueueReceive()** amb un valor de 0 a l'últim paràmetre, que és el temps en *ticks* que ha d'esperar-se a rebre un valor en el cas que la cua estigui buida (veure Llistat 25.4).

En aquest exemple volem que encara que no hi hagi dada a la cua, la tasca segueixi executant-se. Per saber si la funció de rebre dades ha obtingut una dada nova, cal comprovar si ha retornat pdTRUE[2]. Si retorna pdFALSE[3] és que ha acabat el temps d'espera (en el nostre cas 0 *ticks*) i no ha pogut extreure un nou valor de la cua. Aquest és un exemple d'accés no bloquejant a una cua.

Les cues cal crear-les abans d'utilitzar-les cridant a la funció **xQueueCreate()**. Aquesta funció crea la cua amb la longitud desitjada i amb la capacitat indicada (Llistat 25.5). Usualment això es crea a la funció **main()** o, en tot cas, abans d'engegar el Sistema Operatiu.

[2] Valor lògic "Cert" definit a FreeRTOS
[3] Valor lògic "Fals" definit a FreeRTOS

Llistat 25.4: Part principal de la tasca TaskLedToggle

```
for (;;) {
  /* try to get new delay time from queue */
  if (xQueueReceive(queue_buttons, &recv_delay, (TickType_t ) 0)) {
    my_delay = recv_delay;
  }

  /* wait for m_delay & toggle the LED */
  vTaskDelay(my_delay);
  LedToggle();
}
```

Llistat 25.5: Creació d'una cua

```
/* Create Queue */
queue_buttons = xQueueCreate(QUEUE_LENGTH, sizeof(uint32_t));
```

25.2.2 Enviant múltiples dades per una cua

Quan ens cal enviar diverses dades d'una tasca a una altra (o d'una ISR a una tasca, o d'una tasca a una ISR, etc.) la forma més senzilla de fer-ho és fent servir una estructura per fer un paquet.

Suposem que tenim una tasca que llegeix dades d'un acceleròmetre, que treu dades de 16 bits (uint16_t) per cada un dels eixos (X, Y, Z). Aquestes dades les volem enviar a una segona tasca que fa els càlculs pertinents per l'aplicació i, per tant, posem una cua entre les dues tasques.

Per enviar el triplet de dades d'una tasca a l'altra, podríem muntar una cua on enviéssim les dades per ordre (primer X, després Y, després Z, altre cop X, després Y, etc.). Això ens podria portar problemes si la tasca que rep les dades perd una dada, ja que llavors estaríem confonent una dada per una altra.

L'altra opció podria ser posar tres cues, una per cada coordenada i la tasca consumidora anar llegint de cada una. Això però, sembla que és una mica massa complexitat per un problema senzill.

La millor solució consisteix a preparar una estructura de dades i que sigui aquesta estructura la que s'envia per la cua. Així, les tres dades viatgen juntes i cada una associada al seu camp corresponent. La definició de l'estructura haurà de ser comú a totes les tasques i cues que la facin servir; en un projecte gran caldrà definir-la en un *header* comú per la resta de mòduls del projecte.

Així, podríem definir una estructura tal com es veu al llistat 25.6.

I es crea la cua tal com es veu al Llistat 25.7.

Llistat 25.6: Paquet dins d'estructura

```
struct queue_pkt {
   uint16_t eixX;
   uint16_t eixY;
   uint16_t eixZ;
};
```

Llistat 25.7: Creació de la cua amb un paquet de dades

```
queue_handle = xQueueCreate(QUEUE_LENGTH, sizeof(struct queue_pkt));
```

Llistat 25.8: Rebre un paquet de dades de la cua

```
...
if (xQueueReceive(queue_buttons, &pkt, (TickType_t ) 0)) {
  eixX = pkt.eixX;
}
...
```

Així, la informació que es mourà per la cua serà l'estructura i caldrà agafar cada dada de la mateixa de la forma habitual, tal com es veu al Llistat 25.8.

Al repositori l'exemple FreeRTOS_Queue_2 es fa servir una estructura per passar dades mitjançant una cua.

25.3 Mutex

Quan tenim un recurs, *driver*, memòria compartida, secció crítica o qualsevol altre recurs que només es pot fer servir per una sola tasca a cada moment, cal muntar un mecanisme d'exclusió mútua que ens asseguri que no tindrem cap problema [52, pàgina 244].

Aquest mecanisme és molt similar a un semàfor binari però cal incloure algun mecanisme per prevenir la inversió de prioritats. Aquest mecanisme és el Mutex (d'aquí els ve el nom: *Mutual Exclusion*). Els Mutex implementen un mecanisme d'herència de prioritats de tal manera que si una tasca d'alta prioritat està esperant un Mutex que té una tasca de baixa prioritat, aquesta última veu augmentada la seva prioritat a la mateixa prioritat que la tasca d'alta prioritat mentre té el Mutex per tal que tingui més oportunitat d'alliberar-lo ja que altres tasques amb prioritats entre mig poden bloquejar la tasca de baixa prioritat [4] (veure **Capítol 48 - Inversió de prioritats** més endavant).

Per l'ús correcte d'un Mutex, el que es fa és provar d'agafar el Mutex abans d'entrar a la secció crítica, si es té èxit s'executa el que calgui dins la secció crítica i a continuació s'allibera el Mutex. Com ens podem imaginar, cal que el temps que estem dins una secció crítica sigui el més curt possible. Com la resta de crides d'aquesta mena, el paràmetre *timeout* ens permet seleccionar el temps que la tasca ha d'estar esperant auqe el Mutex estigui disponible (des de 0 fins a temps infinit).

En el cas de FreeRTOS cal primer crear el Mutex i a partir de llavors ja es pot fer servir per part de les tasques. Les tasques poden agafar o donar un Mutex amb les mateixes funcions de manegar semàfors que ja coneixem.

Exemple amb Mútex

Al repositori del curs tenim un exemple on dues tasques fan ús d'un recurs compartit com pot ser la consola de debug (amb el printf) i es comparteix amb un Mutex.

A l'exemple tal com està ara, no està definit la macro USE_MUTEX i el codi no en fa ús. Si executem el codi tal qual està, veurem que la sortida de les dues tasques es barreja ja que no hi ha

[4] Aquest problema és el conegut *inversió de prioritats* que pot donar molts mal de caps si no es detecta a temps

Llistat 25.9: Exemple d'ús de macros en C

```
#ifdef USE_MUTEX
  if (xSemaphoreTake(example_mutex, portMAX_DELAY)) {
#else
  {
#endif
```

Llistat 25.10: Sortida de la consola sense Mutex

```
from Task1
Other text Some text from Task 2
from Task1
Other text Some text from Task 2
from Task1
Other text Some text from Task 2
from Task1
```

cap control de qui escriu i quan (Llistat 25.10).

Un exemple d'ús de macros en C es veu al llistat 25.9. En aquest codi, si la macro **USE_MUTEX** no està definida no s'executa la línia 2 i no es té en compte el Mutex.

Si traiem el comentari i activem la macro USE_MUTEX llavors el codi de manegar el Mutex s'activa i llavors veurem que la sortida per la consola ja és la correcta (Llistat 25.11).

Què passa en aquest cas? Doncs que abans de treure el text per la consola, es demana el Mutex i queda protegida la secció crítica i tot funciona com ha de ser.

A l'exemple es fa servir la comanda **taskYIELD()** entre mig dels dos printf per simular que la tasca en aquell punt perd l'execució. Com segur que saps, les condicions de carrera (*race conditions*) són molt complicades de trobar i provocar perquè són infreqüents i només passen de tant en tant; i és per això que provoquem el canvi de tasca amb la comanda **taskYIELD()**.

En aquest exemple i per fer-ho senzill, la crida per demanar el Mutex porta com a segon paràmetre **portMAX_DELAY**, que fa que la tasca quedi bloquejada fins que s'alliberi el Mutex. També es pot afegir un temps d'espera (*timeout*) i llavors la funció retorna quan s'ha agafat el Mutex (i retorna pdTRUE) o quan ha passat el temps d'espera (i retorna pdFALSE).

L'ús de Mútex és necessari per controlar l'accés a qualsevol secció crítica que tinguem al nostre

Llistat 25.11: Sortida de la consola amb Mutex

```
Some text from Task1
Other text from Task 2
Some text from Task1
Other text from Task 2
Some text from Task1
Other text from Task 2
```

projecte. Habitualment en tindrem per cada ús o crida a un *driver* que pugui portar-nos problemes d'aquesta mena. Per exemple, si dues tasques han d'accedir al bus I2C per accedir a diferents sensors caldrà protegir amb un Mutex les crides a les llibreries del sistema.

> **Exercici 25.2** La prioritat de les dues tasques a l'exemple és la mateixa. Com exercici es pot provar de canviar les prioritats i treure els Mutex, a veure què passa i intentar entendre el perquè.
> ■

25.4 *Event Groups*

A més dels mecanismes ja explicats, i que son els més típics en la comunicació i sincronització entre tasques, FreeRTOS ens proporciona un mecanisme que permet bloquejar o desbloquejar una o vàries tasques segons succeeixi un o varis esdeveniments [52, pàgina 266]. Aquest mecanisme es diu *Event Groups* i permet que una o vàries tasques defineixen a quins esdeveniments son sensibles i el *kernel* gestiona tot el necessari. Aquest mecanisme permet:

- Que una tasca o més tasques estiguin bloquejades esperant diversos esdeveniments
- Que un esdeveniment desbloquegi vàries tasques de forma simultània.

Aquest mecanisme es basa en declarar *flags* per cada esdeveniment per tot seguit definir un grup de *flags* anomenat *Event Groups*. Aquests *flags* individuals s'emmagatzemen com un sol bit, de manera que només poden tenir el valor 0 o 1, on el 0 vol dir que l'esdeveniment no ha succeït i l'1 que l'esdeveniment si que ha succeït. Tots aquests bits es guardaran en una variable de 8 o 24 bits depenent com estigui definida la constant **configUSE_16_BIT_TICKS** al fitxer *FreeRTOSConfig.h* (si la constant val 1 el grup conté només 8 bits, si val 0 (valor per defecte) el grup conté 24 bits).

Les tasques que han d'estar pendents d'aquest grup notifiquen al *kernel* de quins esdeveniments particulars volen dependre (bits individuals de la paraula), si s'ha de complir tots o només un (operació "AND" o "OR"), si s'han de netejar els *flags* que han desbloquejat la tasca i el temps que es pot esperar a que això succeeixi (*timeout*).

Així, i de forma general, tindrem que qui manegui els esdeveniments (típicament ISRs, però poden ser altres tasques) posaran els *flags* individuals a '1' quan aquests succeeixin mentre que una o més tasques estaran bloquejades esperant que una sèrie de bits del grup s'activin (només un o tots depenent de la configuració particular).

25.4.1 Exemple de *event groups*

El codi d'aquest exemple es troba al repositori i conté una sola tasca esperant per que es compleixin dos esdeveniments, que serà la pulsació dels dos botons de la placa de prototipat. Per tant, l'exemple farà *toggle* del LED quan s'hagin pres els dos botons (no cal que sigui simultàniament).

Al Llistat 25.12 es presenta la tasca, que simplement es queda bloquejada esperant pels dos bits del grup prèviament definits. Els dos paràmetres següents (tots dos **pdTRUE**) indiquen que cal netejar els esdeveniments i cal esperar a tots els esdeveniments s'hagin activat. Per últim, també es configura un *timeout* infinit perquè la tasca es quedi bloquejada per sempre esperant als dos esdeveniments.

El codi de la ISR (veure Llistat 25.13) conté el codi ja conegut per netejar els flags del mòdul GPIO i tot seguit es notifica l'esdeveniment corresponent al grup amb la funció *xEventGroupSetBitsFromISR*. En aquest cas els paràmetres son: el grup a notificar, el bit del grup a notificar i l'últim paràmetre serveix per rebre la informació de si cal notificar al *kernel* que una tasca s'ha desbloquejat.

Llistat 25.12: Tasca esperant per un grup d'esdeveniments

```
#define FIRST_BUTTON_BIT (1 << 0)
#define SECOND_BUTTON_BIT (1 << 1)

static void TaskLedToggle(void *pParameter) {
  (void) pParameter;

  while (true) {
    xEventGroupWaitBits(event_group, (FIRST_BUTTON_BIT | SECOND_BUTTON_BIT),
        pdTRUE, pdTRUE, portMAX_DELAY);
    LedToggle();
  }
}
```

Llistat 25.13: ISR notificant un esdeveniment a un grup

```
void GPIO_ODD_IRQHandler(void) {
  uint32_t aux;
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* clear flags */
  aux = GPIO_IntGet();
  GPIO_IntClear(aux);

  xEventGroupSetBitsFromISR(event_group, SECOND_BUTTON_BIT, &
      xHigherPriorityTaskWoken);

  /* Awake a task ? */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Llistat 25.14: Tasca esperant un grup d'esdeveniments (OR)

```
static void TaskLedToggle(void *pParameter) {
  (void) pParameter;

  while (true) {
    xEventGroupWaitBits(event_group, (FIRST_BUTTON_BIT | SECOND_BUTTON_BIT),
        pdTRUE, pdFALSE, portMAX_DELAY);
    LedToggle();
  }
}
```

Si a la tasca es canviés la crida per la del Llistat 25.14 llavors la tasca es desbloquejarà qua passi qualsevol dels dos esdeveniments (es farà una "OR" entre els tots els bits del grup enlloc d'una "AND").

25.4.2 Sobre el determinisme dels grups d'esdeveniments

La implementació d'aquests *Event Groups* a FreeRTOS es fan mitjançant una tasca auxiliar del mòdul de Timers software ja que la resolució d'un grup d'esdeveniments no és determinista (ja que no es pot saber per avançat quantes tasques o quants esdeveniments estan involucrats a cada moment). La tasca auxiliar es crea automàticament el primer cop que es crea un grup d'esdeveniments amb la prioritat per defecte d'aquest tasca.

A més, la comunicació entre les funcions de l'API (*xEventGroupSetBitsFromISR()*, etc.) i la tasca auxiliar es fa mitjançant una cua, de manera que no és possible el determinisme de tot el mecanisme pel cas general.

Per tot això, cal ser curós en quins casos fer servir aquest mecanisme, per senzill que pugui semblar i depenent de la criticitat de l'aplicació i del cas particular.

25.5 Conjunt de cues *Queue Sets*

Hi ha casos on una tasca pot voler rebre dades de cues diferents on, potser, per cada cua es reben dades de tipus diferents. FreeRTOS ens proporciona un mecanisme per poder rebre de vàries cues d'una forma senzilla.

El mecanisme és el *Queue set* (conjunt de cues) que permet agrupar tot de cues i semàfors i després consultar per part d'una tasca si hi ha alguna dada disponible a alguna de les cues o semàfor.

El que cal fer és crear un conjunt, afegir-hi els mecanismes de sincronització que es vulguin incloure i ja només cal consultar la disponibilitat a través del conjunt enlloc de cada mecanisme per separat.

Anem a veure-ho en un exemple, que com sempre està al repositori. A l'exemple es modifica l'exemple de cues anterior perquè cada ISR enviï la seva dada per una cua diferent (*queue_buttons_1*, *queue_buttons_2*).

El primer que cal fer és crear el conjunt de cues (*Queue Sets*) tal com es veu al Llistat 25.15. Cal veure que quan és crea el conjunt cal especificar la longitud de totes les cues i semàfors que s'agrupen. Per les cues cal sumar el nombre d'elements, pels semàfors binaris la longitud és 1 i pels semàfors comptadors la longitud és el valor màxim que poden tenir.

A la tasca s'ha canviat com es rep les dades de cada ISR i ara es consulta el conjunt de cues (Llistat 25.16). Aquesta funció retorna el *handler* al mecanisme que té una dada disponible, de

Llistat 25.15: Creació del conjunt de cues

```
...
#define QUEUE_LENGTH (10)

QueueSetHandle_t queue_set;
QueueHandle_t queue_buttons_1;
QueueHandle_t queue_buttons_2;

void main() {
  ...
  queue_set = xQueueCreateSet( QUEUE_LENGTH + QUEUE_LENGTH );
  queue_buttons_1 = xQueueCreate(QUEUE_LENGTH, sizeof(uint32_t));
  queue_buttons_2 = xQueueCreate(QUEUE_LENGTH, sizeof(uint32_t));
  ...
}
```

Llistat 25.16: Tasca que fa servir el conjunt de cues

```
static void TaskLedToggle(void *pParameter) {
  ...
  for (;;) {
    selected_queue = xQueueSelectFromSet(queue_set, 0);
    if ( selected_queue == queue_buttons_1) {
      xQueueReceive(queue_buttons_1, &my_delay, (TickType_t ) 0);
    } else if (selected_queue == queue_buttons_2) {
      xQueueReceive(queue_buttons_2, &my_delay, (TickType_t ) 0);
    }
    vTaskDelay(my_delay);
    LedToggle();
  }
}
```

manera que a continuació es consulta al mecanisme i s'adquireix la dada rebuda. La resta del mecanisme és força similar a l'exemple anterior.

Cal fer notar que l'exemple es fa amb només dues cues, però els *Queue Sets*, malgrat el nom, també poden incloure semàfors de la mateixa manera i consultar-los de la mateixa forma.

25.6 Notificacions a tasques

Les notificacions a tasques (en anglès *Direct to Task Notifications*) son un mecanisme propi de FreeRTOS similar a les cues, semàfors i mútex però més simple i, en alguns casos, més eficient (aquest mecanisme pot ser fins a un 45% més ràpid que un mecanisme basat en un semàfor binari).

Si bé els mecanismes introduïts fins ara eren objectes que existien entre les tasques que comunicaven, les notificacions es fan de forma directe entre ISR i tasques o entre dues tasques sense cap objecte addicional. Això te l'avantatge que s'estalvia memòria, ja que no cal mantenir tanta informació i que el mecanisme és més ràpid, però comporta certes limitacions:

- No es pot enviar una notificació cap a una ISR, tot i que si que es pot a l'inversa.
- Només es pot notificar a una sola tasca, ja que es notifica directament la tasca, no cap mecanisme intermig.

Taula 25.1: Crides de notificacions de tasques i els mecanismes equivalents

Semàfor binari	xTaskNotifyGive() / ulTaskNotifyTake()
Semàfor comptador	xTaskNotifyGive() / ulTaskNotifyTake()
Grup d'esdeveniments	xTaskNotify() / xTaskNotifyWait()
Cua (d'un sol element)	xTaskNotify() / xTaskNotifyWait()

Llistat 25.17: Tasca que espera la notificació

```
static void TaskLedToggle(void *pParameter) {
  (void) pParameter;

  while (true) {
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    TriggerToggle();
    LedToggle();
  }
}
```

- No es pot emmagatzemar dades, ja que el mecanisme de notificació pot manegar una i només una dada.

Les funcions de notificació a tasques necessiten conèixer el *handler* de la tasca a enviar, cosa que s'acostuma a fer mitjançant variables globals.

A la Taula 25.1 es resumeixen les crides a les notificacions i a quins mecanismes poden substituir [5].

25.6.1 Exemple de notificació directa a tasques

Al repositori hi ha l'exemple més senzill on es notifica una tasca que es suspèn esperant un esdeveniment. Aquest esdeveniment ve donat per la pulsació d'un dels botons i la notificació per part de la ISR.

El Llistat 25.17 mostra la tasca, que simplement espera agafar la notificació amb la funció **ulTaskNotifyTake()** per tot seguit fer *toggle* del LED. Els paràmetres de la crida fan que es netegi el flag un cop s'ha rebut i s'esperi indefinidament.

Cada una de les ISR tant sols notifica la tasca amb la crida corresponent, tal com es veu al Llistat 25.18. En aquest cas s'ha de fer servir la variable *taskhandle* que emmagatzema el *handle* a la tasca que està esperant la notificació. Aquesta variable és global a tot el codi i està definida al principi del fitxer main.c.

[5] També hi ha les funcions per ISRs vTaskNotifyGiveFromISR() i xTaskNotifyFromISR()

Llistat 25.18: Tasca que espera la notificació

```
void GPIO_ODD_IRQHandler(void) {
  uint32_t aux;
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;

  /* clear flags */
  aux = GPIO_IntGet();
  GPIO_IntClear(aux);

  vTaskNotifyGiveFromISR(taskhandle, &xHigherPriorityTaskWoken);

  /* Awake a task ? */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

25.7 Comparant temps de resposta

En aquesta secció valorarem els temps de resposta mesurats en els tres mecanismes de comunicació entre esdeveniments i tasques, que son els semàfors, els grups d'esdeveniments i les notificacions directes.

A la Taula 25.2 es resumeixen aquestes mesures i a les Figures 25.3,25.4 i 25.5 es veuen les mesures fetes amb l'oscil·loscopi. Com es pot comprovar, el mecanisme més senzill i ràpid és el de la notificació directa, el semàfor afegeix una mica més de complexitat i per tant de temps de resposta i, per últim, el mecanisme més complex del grup d'esdeveniments és el mecanisme més lent amb diferència.

Cal fer notar també que la resposta és de l'ordre de microsegons, que està per sota del temps de *tick*, cosa que significa que la tasca que s'estigui executant en el moment d'ocorre l'esdeveniment s'interromp i es passa a executar la tasca que està esperant el mecanisme associat a l'esdeveniment.

Taula 25.2: Crides de notificacions de tasques i els mecanismes equivalents

Mecanisme	Temps (microsegons)
Semàfor binary	93,6
Notificació directa	79,2
Grup d'esdeveniments	306

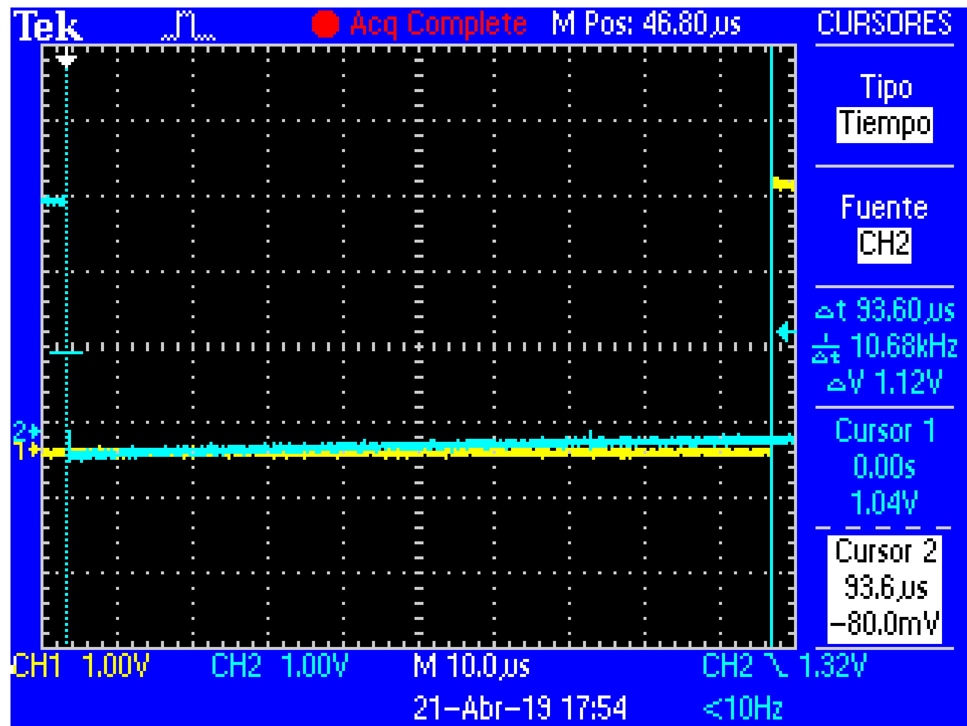


Figura 25.3: Temps de resposta usant semàfors

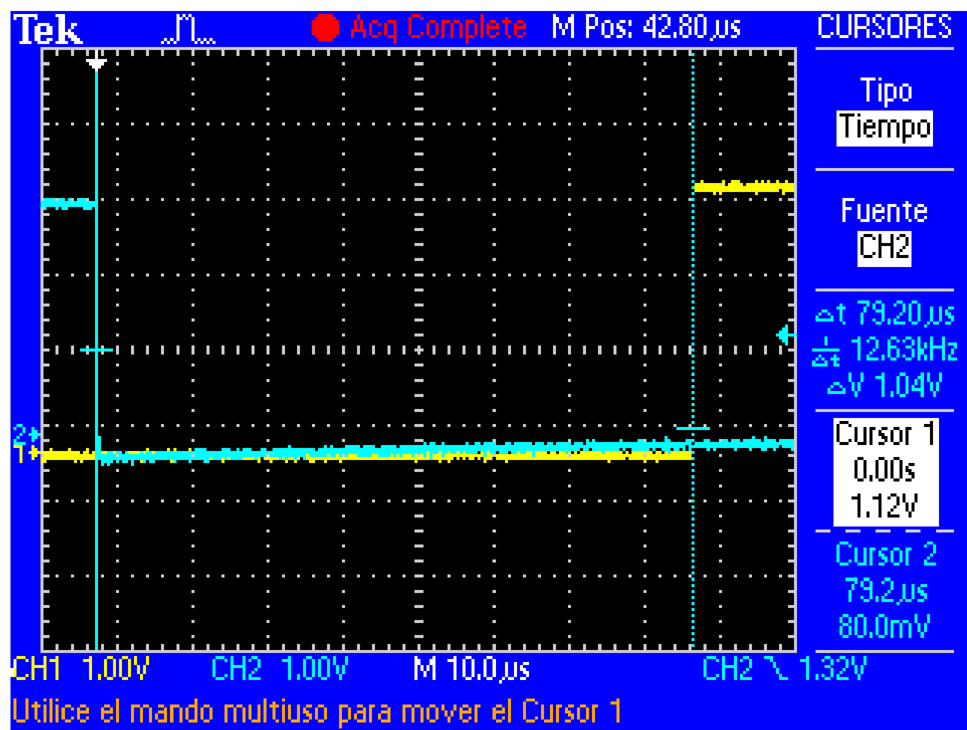


Figura 25.4: Temps de resposta usant notificació a tasca

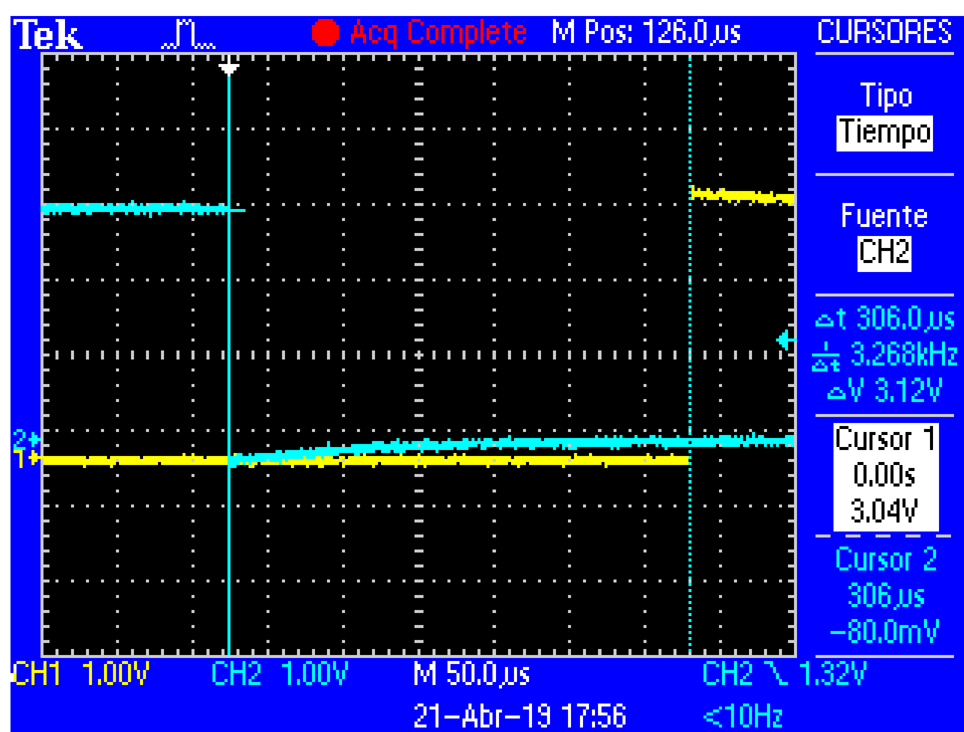


Figura 25.5: Temps de resposta usant groups de notificació

Aquesta pàgina està en blanc expressament, tot va bé.

26. Exemple amb la UART i interrupcions

A l'exemple Freertos_UART hi ha el mateix exemple vist a **Secció 14.3 - Un exemple amb la UART més complicat** però en aquest cas usant FreeRTOS. Per això hi ha uns pocs canvis:

- A les interrupcions de la USART (**USART1_TX_IRQHandler()** i **USART1_RX_IRQHandler()**) se les canvia la prioritat, ja que a FreeRTOS la prioritat de les interrupcions han de tenir un valor diferent a 0. Veure aquest enllaç de la documentació de FreeRTOS [55].
- Enlloc de fer servir el buffer circular es fa servir una cua de FreeRTOS. Així la ISR de recepció guarda el valor rebut a una cua i la ISR de transmissió va buidant la mateixa cua.
- La funció **USART_Send()** també fa servir la cua de transmissió per extreure els caràcters a enviar per la UART
- Enlloc d'un *while(1)* al **main()**, s'ha creat una tasca que prova de llegir un caràcter de la cua de recepció per fer la seva feina.

Llistat 26.1: ISR de RX de la UART amb FreeRTOS

```
void USART1_RX_IRQHandler(void) {
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;
  char data;

  if (USART1->IF & LEUART_IF_RXDATAV) {
    data = USART_Rx(USART1);
    xQueueSendFromISR(USART_RX_queue, &data, &xHigherPriorityTaskWoken);
    USART_IntClear( USART1, USART_IEN_RXDATAV);
  }

  /* Awake a task ? */
  portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

Llistat 26.2: ISR de TX de la UART amb FreeRTOS

```
void USART1_TX_IRQHandler(void) {
  char data;

  USART_IntClear( USART1, USART_IEN_TXC);
  if (xQueueReceiveFromISR(USART_TX_queue, &data, 0) == pdTRUE) {
    USART_Tx(USART1, data);
  }
}
```

Llistat 26.3: funció UART_Send() per FreeRTOS

```
void USART_Send(USART_TypeDef *usart) {
  int data;

  if (xQueueReceive(USART_TX_queue, &data, 0) == pdTRUE) {
    USART_Tx(USART1, data);
  }
}
```

Llistat 26.4: Tasca principal de l'exemple

```
static void UARTTask(void *pParameter) {
  (void) pParameter;
  char recv_char;
  char tx_char;

  for (;;) {
    if (xQueueReceive(USART_RX_queue, &recv_char, portMAX_DELAY)) {
      tx_char = recv_char;
      xQueueSend(USART_TX_queue, &tx_char, 0);
      tx_char++;
      xQueueSend(USART_TX_queue, &tx_char, 0);
      tx_char++;
      xQueueSend(USART_TX_queue, &tx_char, 0);
      USART_Send(USART1);
    }
  }
}
```

Per la resta, l'aplicació té el mateix funcionament que l'exemple sense RTOS. Es pot consultar l'exemple original per una millor explicació del funcionament (veure **Secció 14.3 - Un exemple amb la UART més complicat**).

Aquesta pàgina està en blanc expressament, tot va bé.

27. Una aplicació completa amb FreeRTOS

Per resumir i posar un exemple de tot el vist sobre FreeRTOS, agafarem l'aplicació d'exemple feta a **Capítol 21 - Una aplicació completa** i es transformarà en una aplicació amb FreeRTOS. Per començar es treballarà amb la versió amb *polling* de l'aplicació original.

27.1 Tasques

Una de les característiques d'un RTOS és que les diferents tasques a fer per l'aplicació es divideixen en tasques. En aquest cas, sembla senzill pensar que es pot dividir la feina a fer en dos parts: (i) llegir la dada del sensor i (ii) canviar el duty cycle del PWM segons el valor llegit. En resum, les tasques seran:

- **ReadSensor_task**: aquesta tasca llegeix periòdicament el valor de proximitat del sensor i envia aquesta dada cap a l'altra tasca. Això ho fa cada mig segon i fa *polling* del registre d'*estatus* del sensor (veure llistat 27.1).
- **MngData_task**: aquesta tasca rep la dada de proximitat i fa dues coses: la treu per la consola de *debug* amb un **printf()** i canvia el ritme de pampallugueig del LED segons aquest valor. Aquesta tasca es bloqueja esperant obtenir una dada, i en quan en rep una canvia el paràmetre del PWM i imprimeix el valor per la consola (veure llista 27.2).

Les dues tasques s'han de poder comunicar, doncs la tasca que llegeix el valor de proximitat del sensor l'ha de poder enviar a la tasca que gestiona el PWM. Així doncs, tenim dues tasques comunicades amb una cua (anomenada **sensor_data_queue**). Com que el tipus de dades que hem d'enviar d'una tasca a l'altra és tant sols el valor de proximitat proporcionat pel sensor, la cua pot emmagatzemar directament aquest valor. És per això que la cua es crea amb 4 posicions de 8 bits cada una (veure Llistat 27.3).

Com ja hem vist anteriorment, es creen les dues tasques dins de la funció **main()** (Llistat 27.3).

Llistat 27.1: Tasca ReadSensor

```
static void ReadSensor_task(void *pParameter) {
  uint8_t p_data;
  bool ret;

  (void) pParameter;
  APDS_9960_InitProximity();

  while (pdTRUE) {
    ret = APDS_9960_ReadProximity(&p_data);

    if (ret == true) {
      xQueueSend(sensor_data_queue, &p_data, portMAX_DELAY);
    }

    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}
```

Llistat 27.2: Tasca MngData

```
static void MngData_task(void *pParameter) {
  uint8_t p_data;

  (void) pParameter;

  while (pdTRUE) {
    xQueueReceive(sensor_data_queue, &p_data, portMAX_DELAY);
    printf("Proximity: %d\r\n", p_data);

    /* Convert from range 0 - 256 to 0 - 100 */
    PWM_Set((p_data * 100) / 256 );
  }
}
```

Llistat 27.3: Creació de tasques

```
main() {
  ...
  /* Create queue to send data between two tasks */
  sensor_data_queue = xQueueCreate(4, sizeof(uint8_t));

  /* Create read sensor task */
  xTaskCreate(ReadSensor_task, (const char *) "ReadSensor",
          configMINIMAL_STACK_SIZE-65, NULL, READ_TASK_PRIORITY, NULL);

  /* Create print & LED ctrl task */
  xTaskCreate(MngData_task, (const char *) "MngData",
          100-5, NULL, MNG_TASK_PRIORITY, NULL);
  ...
```

27.2 Modificant el *wrapper* d'I2C

Com que FreeRTOS és un sistema operatiu preemptiu, cal que les funcions de les biblioteques es puguin fer servir per diverses tasques alhora (siguin re-entrants) [52, pàgina 236]. Habitualment això es fa amb un *mutex* que protegeixi la o les seccions crítiques de cada biblioteca. En el cas de la biblioteca *I2C_Wrapper* es fa amb un sol *mutex* que protegeix la crida a la transferència I2C pròpiament dita (veure Llistat 27.4).

D'aquesta manera en el cas que dues tasques fessin servir el *wrapper* per accedir al bus I2C, quan cridessin a la funció **I2C_WriteRegister()** o **I2C_ReadRegister()** aquestes funcions es protegeixen de la re-entrada amb el mutex *I2C_mutex* impedint que es pogués cridar dos cops (un cop de cada tasca) a la funció **I2C_Transfer()**, que faria que les transferències I2C no es fessin correctament.

El *mutex* (anomenat **I2C_mutex**) està definit com *static* dins el fitxer **I2C_Wrapper.c**. Això farà que aquesta variable només estigui disponible dins el mòdul i no sigui una variable global a tot el projecte. Aquest *mutex* s'inicialitza a la funció **I2C_initialize()**.

La resta de biblioteques usades no cal canviar-les respecte a l'aplicació *baremetal*, ja que la biblioteca **APDS-9960** fa servir la biblioteca **I2C** que ja està preparada per ser re-entrant i la biblioteca **BSP** no necessita de cap canvi perquè funcioni sota FreeRTOS ja que no fa ús de cap recurs compartit ni cal protegir les funcions per la seva re-entrada.

Llistat 27.4: Part de la funció I2C_WriteRegister() adaptada a FreeRTOS

```
bool I2C_WriteRegister() {
  ...
  xSemaphoreTake(I2C_mutex, portMAX_DELAY);

  I2C_Status = I2C_TransferInit(I2C0, &seq);

  while (I2C_Status == i2cTransferInProgress) {
    I2C_Status = I2C_Transfer(I2C0);
  }

  xSemaphoreGive(I2C_mutex);
  ...
}
```

```
static SemaphoreHandle_t I2C_mutex;
```

```
I2C_initialize() {
  ...
  I2C_mutex = xSemaphoreCreateMutex();
  ...
}
```

27.3 Analitzant les diferències

Un factor a tenir en compte quan treballem amb RTOS és la sobrecàrrega que provoquen. Aquest sobre-preu pot ser en codi i complexitat del mateix, en quantitat de memòria utilitzada o en la complexitat intrínseca de fer-los servir.

Anem a comprovar primer el sobrecost en l'ús de la memòria, ja que acostuma a ser el recurs més escàs en un sistema encastat.

Mirant la Taula 35.1 veure que l'aplicació amb FreeRTOS necessita més quantitat de memòria tant FLASH com RAM. Em quantitat de codi és evident, ja que hi hem afegit tot el codi del S.O. Pel que a la memòria RAM (seccions **data** i **bss**) augmenta considerablement l'ús de la secció **bss**. Aquesta secció la utilitza el FreeRTOS per reservar-la per l'*stack* de cada una de les tasques. Com que aquest regió es reserva de forma estàtica ja apareix a la comanda *size*.

R Recordem que **text** és l'espai de memòria FLASH necessari; **data** la quantitat de bytes de variables inicialitzades (ocupen tant FLASH com RAM) i **bss** la quantitat de memòria RAM de la que cal disposar per variables (veure **Subsecció 2.3.2 - Mida del codi i seccions de memòria**).

Taula 27.1: Ocupació de memòria de les dues aplicacions

Aplicació	text	data	bss
Baremetal_App_1	13126	120	68
FreeRTOS_App_1	22884	124	2368

28. Ús del watchdog en RTOS

Quan es treballa en un entorn amb un RTOS, cal estudiar bé com fer servir el *watchdog*. La primera pensada pot ser d'afegir les crides per alimentar el *watchdog* a cada una de les tasques com si fossin mini-aplicacions individuals. Aquesta aproximació, però, faria que el sistema mai es reiniciï encara que una tasca deixi de funcionar o tingui algun problema greu, ja que la resta de tasques seguirien alimentant-lo.

La solució més habitual és la de tenir una tasca dedicada a alimentar el *watchdog* i que rebi una mena d'OK de cada una de les tasques restants del sistema. D'aquesta forma, si una tasca deixa de funcionar, aquesta tasca dedicada ho detectarà i deixarà d'alimentar el *watchdog* provocant que el sistema es reiniciï (Llistat 28.1).

El codi que es veu al Llistat 28.1 proporciona la funció **watchdogTouch()** que és la que haurà de cridar les diferents tasques del sistema, cadascuna amb un paràmetre **WATCHDOG_TASK<N>** diferent i únic.

Com es pot veure a l'exemple, la variable local a la biblioteca *watchdog_list* emmagatzema l'estat de totes les tasques i s'hi accedeix a la funció **watchdogTouch()** que protegeix l'accés amb un *mutex*. La tasca **watchdogTask()** avalua aquesta variable d'estat i si tot ha anat correctament (totes les tasques han cridat la seva funció almenys un cop), alimenta el *watchdog*. En cas contrari, la tasca no l'alimenta i acabarà per reiniciar el sistema.

A l'exemple aquesta tasca s'executa un cop cada segon, i el *watchdog* s'ha de configurar d'acord a aquest temps (un temps de *watchdog* de 2 segons seria l'adequat). La resta de tasques haurien de cridar la funció **watchdogTouch()** amb un període de temps prou curt (per exemple cada 500 mil·lisegons) per tal de que tot el sistema s'executi correctament.

Llistat 28.1: Codi d'exemple de la tasca de control del **watchdog**

```
#define WATCHDOG_TASK1 0x01
#define WATCHDOG_TASK2 0x02
#define WATCHDOG_TASK3 0x04
#define WATCHDOG_TASK4 0x08
#define WATCHDOG_FULL  0x0F

static uint8_t watchdog_list;
SemaphoreHandle_t watchdog_mutex;

void watchdogTouch(uint8_t task) {
  xSemaphoreTake(watchdog_mutex, portMAX_DELAY);
  watchdog_list |= task;
  xSemaphoreGive(watchdog_mutex);
}

void watchdogClear() {
  xSemaphoreTake(watchdog_mutex, portMAX_DELAY);
  watchdog_list = 0;
  xSemaphoreGive(watchdog_mutex);
}

void watchdogTask(void *parameter) {
  ...
  WDOG_Init(&init);
  watchdog_mutex = xSemaphoreCreateMutex();

  watchdog_list = 0;
  while(1) {
    if (watchdog_list == WATCHDOG_FULL) {
      WDOG_Feed();
      watchdogClear();
    }

    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}
```

29. *Drivers* en multi-tasca

Quan fem servir un dispositiu (I2C, SPI, etc.) en un entorn multi-tasca com és FreeRTOS podem tenir el problema de dos o més tasques accedint simultàniament a un mateix recurs (el mòdul hardware del microcontrolador). És per això que cal escriure els drivers per accedir a dispositius d'una manera especial quan treballem en entorns multi-tasca.

Ens podem imaginar què passaria si dues tasques intentessin accedir al bus I2C alhora? Què passaria quan una estigues llegint pel bus i, pel que fos, quedés suspesa i la següent tasca a executar-se comences una transferència d'escriptura pel mateix bus? Segurament es corromprien totes dues transferències o s'estarien fent transferències errònies al sistema.

El que volem evitar és que dues o més tasques facin ús alhora del recurs compartit. Per tant, caldrà establir un control d'accés de manera que fins que una tasca no ha acabat de fer servir el recurs l'altra tasca s'ha d'estar esperant. Ja hem vist una aproximació senzilla a **Secció 27.2 - Modificant el *wrapper* d'I2C**, però ara anem a mirar-nos-ho amb més deteniment.

Hi ha diferents maneres de fer això, aquí farem servir la més senzilla i estesa, que és escriure un wrapper (embolcall) que protegeixi les funcions del driver i que seran les que farem servir a les nostres tasques. Aquest wrapper contindrà totes les funcions necessàries i les protegirà amb un mutex (veieu Fent servir Mutex en aquest mateix curs). Aquest mutex ens servirà per controlar l'accés a les parts compartides, que seran les pròpies crides al driver del més baix nivell.

Veiem-ho amb un exemple fent un *wrapper* al driver d'I2C de Silicon Labs que tenim a una aplicació completa (codi al github). En el cas que tinguéssim un sistema on hi hagués més d'un dispositiu Slave connectat el bus, que hi accedeixen dues tasques diferents, podriem trobar-nos amb el problema que comentàvem d'accés múltiple. Per tant, ens cal protegir els accessos amb el mutex tal com hem comentat.

El primer que caldrà és definir una funció d'inicialització del wrapper I2C, que podria quedar com es veu al Llistat 29.1.

La funció tant sols crea un mutex i inicialitza el driver de la biblioteca emlib d'I2C del fabricant. El

Llistat 29.1: Inicialització del *wrapper* I2C amb Mutex

```
/****************************/
/*   Fitxer I2C_Wrapper.h   */
/****************************/
typedef struct I2C_Handle_t* I2C_WrapperHandler_t;

I2C_WrapperHandler_t I2C_initialize(void);
bool I2C_WriteRegister(I2C_WrapperHandler_t handlr, uint8_t addr, uint8_t reg,
    uint8_t data);
bool I2C_ReadRegister(I2C_WrapperHandler_t handlr, uint8_t addr, uint8_t reg,
    uint8_t *val);

/****************************/
/*   Fitxer I2C_Wrapper.c   */
/****************************/
struct I2C_Handle_t {
   SemaphoreHandle_t mutex;
};

static struct I2C_Handle_t i2c_hdnl = {0};

I2C_WrapperHandler_t wrapper_I2C_Init() {

  if (i2c_hdnl.mutex == NULL) {
    i2c_hdnl.mutex = xSemaphoreCreateMutex();

    I2C_Init (...);
  }

  return &i2c_hdnl;
}
```

Llistat 29.2: Modificacions a les funcions *wrapper* I2C amb Mutex

```
bool wrapper_I2C_ReadReg(I2C_WrapperHandler_t handlr, uint8_t address, uint8_t
    reg, uint8_t *data) {

  xSemaphoreTake (handlr->mutex, portMAX_DELAY);
  ...
  I2C_Transfer( ... );
  ...
  xSemaphoreGive (handlr->mutex);

}

bool wrapper_I2C_WriteReg(I2C_WrapperHandler_t handlr, uint8_t address, uint8_t
     reg, uint8_t data) {
  xSemaphoreTake (handlr->mutex, portMAX_DELAY);
  ...
  I2C_Transfer ( ... );
  ...
  xSemaphoreGive (handlr->mutex);
}
```

mutex el retorna com un tipus handler de l'I2C (I2C_WrapperHandler_t) i serà el primer paràmetre que caldrà passar a la resta de crides a les funcions del wrapper.

Així, podem modificar les dues funcions per accedir al bus I2C i que provin d'accedir al mutex, les modificacions podrien quedar tal com es veu al Llistat 29.2.

Així, amb aquests canvis el que tenim ara és que una funció d'accés al bus I2C no es col·lissonarà amb una altra, ja que abans d'intentar accedir-hi haurà d'agafar el mutex. Si no ho aconsegueix, la funció es queda esperant-lo un temps infinit (es bloqueja la funció i la tasca que l'hagi cridada). Quan estarà disponible el mutex? Doncs quan una altra funció d'una altra tasca acabi el seu accés i alliberi el bus.

Cal veure també que el tipus del handler (I2C_WrapperHandler_t) és l'únic tipus que és públic del mòdul i així amaguem l'implementació de l'estructura del handler. En aquest cas el handler és una estructura amb només un mutex, però si més endavant cal afegir-hi més informació no farà que canviï el tipus del handler que fan servir els diferents mòduls.

Aquesta és una bona pràctica per amagar l'implementació de la definició i deixant independent una de l'altra i donant-los la llibertat de canviar l'estructura sense haver de canviar res del codi que fa servir la biblioteca.

També cal veure que el handler és, de fet, un apuntador a una estructura. Això també és una pràctica comuna, ja que és molt més ràpid i eficient passar com a paràmetre un apuntador (que no deixa de ser un tipus de 32 bits) que no pas passar tota l'estructura sencera (que poden ser força camps i molt costosa de passar, copiar, etc.).

Un exemple de canvi a l'estructura del handler podria ser afegir el timeout que volem per provar d'accedir al mutex associat, de manera que a la funció I2C_initialize() se li passés el timeout desitjat i es guardes a l'estructura handler. Els canvis serien els que es veuen al Llistat 29.3.

Aquesta canvis només implicarien afegir el paràmetre de timeout a la crida d'inicialització de l'I2C i cap altre canvi per part dels mòduls que facin servir aquesta biblioteca.

Llistat 29.3: Afegint més dades a l ’estructura del *wrapper* I2C amb mutex

```
struct I2C_Handle_t {
   SemaphoreHandle_t mutex;
   TickType_t timeout;
};

I2C_WrapperHandler_t I2C_initialize(TickType_t timeout) {
  ...
  i2c_hdnl.mutex = xSemaphoreCreateMutex();
  i2c_hdnl.timeout = timeout;
  ...
}

bool I2C_WriteRegister(I2C_WrapperHandler_t handlr, uint8_t addr, uint8_t reg,
    uint8_t data) {
  ...
  xSemaphoreTake(handlr->mutex, handlr->timeout);
  ...
}
...
```

Un altre canvi que es podria afegir és en el cas que tinguem més d’un perifèric del mateix tipus (és a dir, 3 SPIs, o 2 I2C, o...) caldria llavors passar quin dels perifèrics volem inicialitzar i fer servir. Per tant, una opció seria passar com a paràmetre a la funció I2C_initialize() quin dels perifèrics I2C es vol inicialitzar. El handler que tornés hauria de ser diferent en funció del perifèric a treballar i quin és s’hauria de guardar a l’estructura oculta.

Models de programació

30 **Model d'interfície amb perifèrics** **177**

30.1 *Polling* d'esdeveniments

30.2 Interrupcions

31 **Models de computació** **179**

31.1 Bucle de control

31.2 Màquines d'estat finits

31.3 Codificant FSMs

31.4 Flux de dades

32 **Tractament del temps** **191**

32.1 FSMs amb temps

32.2 Tasques periòdiques

32.3 Multitasca

Aquesta pàgina està en blanc expressament, tot va bé.

Fins ara hem vist com controlar els perifèrics més habituals que poden trobar a un microcontrolador, fent ús de les biblioteques dels fabricants. També s'ha observat que la programació per sistemes encastats és força diferent a la d'una aplicació d'escriptori o d'una *app* per un telèfon mòbil. Per això cal ara dedicar uns capítols a presentar els diferents models de programació més utilitzats en la programació de sistemes encastats. També caldrà introduir conceptes teòrics potser nous i classificar aquestes models segons diferents aspectes per donar un seguit de criteris a l'hora de decidir quin model utilitzar per cada aplicació concreta.

Aquesta pàgina està en blanc expressament, tot va bé.

30. Model d’interfície amb perifèrics

Una primera classificació de com es planifica i s’acaba codificant la nostra aplicació és segons com es faci la interfície amb els diversos perifèrics. Aquesta classificació es basa en la ja coneguda dicotomia entre el *polling* i el treball amb interrupcions sobre perifèrics. Aquesta classificació és independent de les següents classificacions que es faran, tal com es veurà més endavant i, de fet, es poden combinar amb les classificacions següents segons les necessitats de cada aplicació.

30.1 *Polling* d’esdeveniments

El primer model de programació i potser el més senzill és el d’un bucle infinit que està en una espera activa (*polling*) de certs esdeveniments per actuar com calgui segons l’aplicació. En aquest model no es fan servir les interrupcions i en tot moment es prova de fer la lectura dels perifèrics o sensors que han de permetre una lectura nova i potser provocar un canvi en el sistema. Per tant, aquest model consistirà bàsicament en un bucle infinit dins la funció *main* de l’aplicació. Aquest bucle anirà repetint indefinidament les lectures necessàries i les actuacions pertinents si s’han pogut fer.

Un exemple d’aquesta mena d’estil de programació s’ha vist a **Capítol 21 - Una aplicació completa**, on l’aplicació ha de llegir un sensor de distància per mostrar-la en un LEDs controlant la seva potència. En aquest exemple, el codi conté una funció **main** que hi ha un bucle infinit on contínuament es prova de llegir el valor de proximitat del sensor i, si és el cas, variar la lluminositat del LED segons la lectura feta (veure també el codi 21.5).

Aquesta mena de programació és suficient per múltiples aplicacions senzilles, on la lògica de l’aplicació depèn de poques variables o condicions i el concepte de *timeout* o de temps en general n’és absent. Si, per exemple, volem que transcorri algun temps entre algunes instruccions o esperar-se un determinat temps per realitzar una operació, caldrà introduir el temps al model.

També cal tenir en compte que aquest model fa que tota l’estona el microcontrolador estigui treballant i, per tant, el consum energètic serà el màxim. Això per múltiples aplicacions no serà cap problema, però s’haurà de tenir en compte en aplicacions orientades al baix consum.

30.2 Interrupcions

L’altre opció és dissenyar l’aplicació de manera que els diversos perifèrics llencin interrupcions quan hagin fet les diverses tasques necessàries i el microcontrolador pugui estar *Idle* tot esperant per rebre les interrupcions i seguir amb l’aplicació.

Normalment aquest mode de treballar fa el codi una mica més complex (veure **Llistat 7.1 - Exemple d’ISR per GPIO** i **el repositori**). En aquests casos es configuren primer tots els perifèrics necessaris per l’aplicació i a partir d’allà es té o bé un bucle infinit molt senzill processant dades o es fa tot en funcions de *callback* i dins les ISR i al final del main tant sols hi ha un bucle infinit sense fer res.

En aquest model, el microcontrolador podrà estar en un mode de baix consum fins que no es generi una interrupció d’algun dels perifèrics, baixant considerablement el consum energètic de tot el sistema (vegeu **Secció 35.3 - Estratègies de baix consum**).

31. Models de computació

Segons com es dissenyi i codifiqui el còmput de les dades d'entrada per obtenir unes sortides determinades, tenim diferents models de computació. Anem a veure els més comuns.

31.1 Bucle de control

Una característica força diferent entre un codi de programa per un sistema encastat respecte a un codi per una aplicació habitual és que en el cas dels encastats el programa no pot acabar mai. I això és perquè el codi de programa és únic, i per tant, si acaba el codi el microcontrolador no tindrà cap altre codi a executar i acabarà per reiniciar-se el sistema (que pot ser el comportament desitjat en algun cas).

Dit això, el cas més senzill d'estil de programació per sistemes encastats és un simple bucle infinit on s'executen les operacions a realitzar per l'aplicació desitjada. Abans d'aquest bucle es configuren i preparen els dispositius i variables necessàries, tal com es pot veure a diferents exemples (GPIO_1, Printf_SWO, PWM_1).

L'exemple **Llistat 6.1 - Codi d'exemple de GPIO al repositori**) és un exemple senzill d'aquest tipus de codi.

Quan l'aplicació es complica i comença a tenir més camins de decisió i condicions, s'acostuma a canviar el model cap a màquines d'estat finits, com es veurà al següent apartat.

Aquest model de programació és el que es fa servir en Arduino, on s'ha separat en dues funcions, una per configurar els perifèrics i demès (funció **setup**() i la funció principal que es va cridant un cop i un altre (funció **loop**(). La funció **main()** del sistema Arduino bàsicament executa el que es veu al llistat 31.1 (es pot veure a $ARDUINO$/hardware/arduino/avr/cores/arduino/main.cpp).

.

Llistat 31.1: funció main() d'Arduino

```
int main(void)
{
  ...

  setup();

  for (;;) {
    loop();
    if (serialEventRun) serialEventRun();
  }
 return 0;
}
```

31.2 Màquines d'estat finits

Una màquina d'estats (FSM[1], de *Finite State Machine* en anglès) és un model de màquina que reacciona a certes entrades i calcula el valor per les sortides, segons l'estat en que estigui. També es pot veure com un seguit d'estats possibles (finit!) en el que pot estar la màquina (el nostre programa) i que va canviant segons les entrades del moment. Les sortides depenen de l'estat on s'està i/o de les entrades (segons el tipus de màquina que s'estigui modelant: màquina de Moore o de Mealy) [56].

R Formalment un FSM es pot definir com una 6-tupla de (S, I, O, F, H, s) tal que:
- S és un conjunt finit d'estats $s_0, s_1, \ldots, s_n$
- I és un conjunt d'entrades $i_0, i_1, \ldots, i_m$
- O és un conjunt de sortides $o_0, o_1, \ldots, o_k$
- F és la funció de transició del tipus: $F : S \times I \rightarrow S$
- H és la funció de sortida tal que:
  - Si la màquina és del tipus Moore: H és del tipus: $H : S \rightarrow O$
  - Si la màquina és del tipus Mealy, H és del tipus: $H : S \times I \rightarrow O$
- $s \in S$ és l'estat inicial

Així, el diagrama d'estats de la FSM que implementaria l'exemple vist a (veure **Llistat 6.1 - Codi d'exemple de GPIO** i **el repositori**) seria el que es veu a la Figura 31.1.

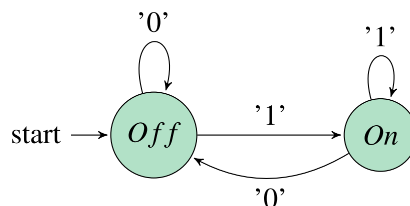


Figura 31.1: FSM per l'exemple GPIO_1

On l'entrada ('1' o '0') es correspon amb l'entrada del pin corresponent, i *On* i *Off* vol dir que en aquell està el LED encès o apagat.

Aquesta FSM es pot codificar de la manera que es veu al Llistat 31.2. En aquest codi es pot veure que dins el el bucle infinit de la funció *main* hi ha una estructura *switch-case* per cobrir tots els

[1] *Finite state machine*

Llistat 31.2: Codi d'exemple de GPIO

```
enum {On, Off} state;
state = On;

int input_value;
...

/* Infinite loop, FSM*/
while (1) {
  input_value = GPIO_PinInGet(gpioPortD, 8);
  switch (state) {
    case On:
        GPIO_PinOutSet(gpioPortD, 7);
        if (input_value == 0) {
            state = Off;
        } else {
            state = On;
        }
        break;
    case Off:
        GPIO_PinOutClear(gpioPortD, 7);
        if (input_value == 1) {
            state = On;
        } else {
            state = Off;
        }
        break;
    case default:
        /* something wrong has happened */
        GPIO_PinOutClear(gpioPortD, 7);
        state = On;
        break;
  }
}
```

estats. Aquests es defineixen amb un *enum* amb els noms desitjats i es crea una variable d'aquest tipus, que s'inicialitza amb l'estat inicial (en aquest cas l'estat **On**). Després, per cada estat s'avalua l'entrada i es pren la definició de quin ha de ser el proper estat. En aquest exemple la FSM és de tipus Moore, i per això la sortida del LED està fixada per cada estat. Si es vol implementar amb una màquina de Mealy el codi hauria de ser tal com es veu al Llistat 31.3, on la sortida depèn de l'estat i l'entrada actual.

Les FSM son una bona manera d'estructura la solució de l'aplicació, ja que cal dissenyar-les i pensar-les abans de començar a escriure codi. L'ús d'aquests mecanismes també ajuda a reduir el nombre de camins d'execució, cosa que simplifica el test del codi.

31.2.1 Màquina d'estats finits estesa

Les màquines d'estats finits esteses (EFSM[2]) amplien el concepte de les FSM amb el de **variables** que mantenen valors interns de manera que la funció de transició pot preguntar per valors d'aquestes variables [57].

[2] *Extended finite state machine*

Llistat 31.3: Codi d'exemple de GPIO

```
...
  switch ( state ) {
    case On:
        if (input_value == 0) {
            GPIO_PinOutClear(gpioPortD, 7);
            state = Off;
        } else {
            GPIO_PinOutSet(gpioPortD, 7);
            state = On;
        }
        break;
    case Off:
        if (input_value == 1) {
            GPIO_PinOutSet(gpioPortD, 7);
            state = On;
        } else {
            GPIO_PinOutClear(gpioPortD, 7);
            state = Off;
        }
        break;
    case default:
        /* something wrong has happened */
        GPIO_PinOutClear(gpioPortD, 7);
        state = On;
        break;
  }
}
```

> **R** Formalment un EFSM es pot definir com una 8-tupla de (S, I, O, D, E, U, F, s) tal que:
> - S és un conjunt finit d'estats $s_0, s_1, ..., s_n$
> - I és un conjunt d'entrades $i_0, i_1, ..., i_m$
> - O és un conjunt de sortides $o_0, o_1, ..., o_k$
> - D és un espai lineal de j dimensions $D_1 \times D_2 \times ... \times D_j$
> - E és un conjunt de funcions d'activació del tipus: $E : D \to 0, 1$
> - U és un conjunt de funcions d'actualització del tipus: $U : D \to D$
> - F és la funció de transició del tipus: $F : S \times I \times E \to S \times U \times O$
> - $s \in S$ és l'estat inicial

Amb aquesta mena de màquines d'estats, es poden tenir variables que, per exemple, comptin fins a un cert valor i llavors permetre un canvi d'estat, o que una variable controli un interval de temps, etc.

31.2.2 Un exemple amb FSM

Veiem un exemple dissenyant un termòstat senzill amb la nostra placa d'avaluació. Es simularà la lectura d'un termòmetre amb el potenciòmetre que ja es va fer servir a l'exemple **Capítol 12 - ADC** i es farà servir un dels LEDs per simular que s'engega l'escalfador d'aigua.

Així, per implementar un termòstat senzill, cal implementar la màquina d'estats de la Figura 31.2. En aquest diagrama d'estats no es dibuixen les transicions que mantenen l'estat, que en aquest cas seria si no es compleixen les condicions de temperatura (si la temperatura és superior a 21°C i està a l'estat Off es manté l'estat, el mateix per l'estat On si la temperatura està per sota del 23°C).

> **R** Que hi hagin dos estats i les transicions entre ells sigui amb temperatures diferents (s'engega a 21°C i s'apaga amb 23°C) serveix per no tenir un termòstat engegant-se i parant-se contínuament un cop s'arriba a la temperatura indicada.

El codi es troba al repositori, al **projecte FSM_1**.

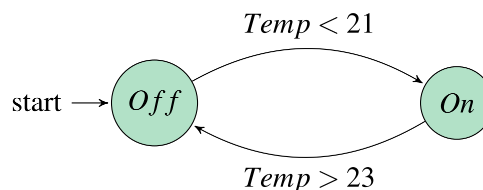


Figura 31.2: FSM d'un termòstat senzill

El codi d'aquest projecte comparteix funcions o crides a la biblioteca ADC d'EMLIB ja vistes a **Secció 12.1 - Exemple d'ADC**. S'han encapsulat dins la funció **ADCGetValue()**, que fa *polling* de l'ADC per obtenir un valor de conversió (veure el Llistat 31.4).

La funció **getTemperature()** utilitza la funció anterior per obtenir un valor de l'ADC, convertir-lo a valor de temperatura (simulada en aquest exemple) i retornar el valor calculat (veure el Llistat 31.5).

També existeixen unes funcions de simulació **SwitchOff()** i **SwitchOn()** que son les que engegarien i pararien l'escalfador d'aigua. Per les proves aquestes funcions encenen o apaguen un dels LEDs de la placa.

La implementació de la FSM és molt senzilla i es veu al codi del Llistat 31.6 i al repositori. Primerament es llegeix la temperatura (simulada) i llavors, segons quin estat estigui la màquina, es compara amb un valor o un altre per saber si cal canviar d'estat o mantenir-se en l'actual. A cada

Llistat 31.4: funció ADCGetValue()

```
static uint32_t ADCGetValue() {
  uint32_t ADCvalue = 0;

  ADC_Start(ADC0, adcStartSingle);
  while (ADC0->STATUS & ADC_STATUS_SINGLEACT);

  ADCvalue = ADC_DataSingleGet(ADC0);

  return ADCvalue;
}
```

Llistat 31.5: funció getTemperature()

```
static uint32_t getTemperature() {
  uint32_t raw_sensor_value;
  uint32_t temperature_value;

  raw_sensor_value = ADCGetValue();

  /* Fake conversion: just for the example we map 0..4095 values of the ADC to
      15..35 celsius degree */
  temperature_value = ((raw_sensor_value * 20) / 4095) + 15;

  return temperature_value;
}
```

Llistat 31.6: Codi de la FSM per un termòstat senzill

```
while (1) {
  temperature = getTemperature();

  switch (state) {
    case Thermo_OFF:
      if (temperature < MIN_TEMPERATURE) {
        state = Thermo_ON;
        printf("Temp: %ld. Changing state to Thermo_ON\r\n",
          temperature);
      }
      SwitchOff();
      break;

    case Thermo_ON:
      if (temperature > MAX_TEMPERATURE) {
        state = Thermo_OFF;
        printf("Temp: %ld. Changing state to Thermo_OFF\r\n",
          temperature);
      }
      SwitchOn();
      break;

    default:
      state = Thermo_OFF;
      SwitchOff();
  }
}
```

estat es crida a la funció de sortida **SwitchOff()** o **SwitchOn()** (la màquina és una màquina d'estats de Moore).

Cal fer notar que la FSM està permanentment consultant el valor del termòmetre (simulat), ja que quan acaba una avaluació de l'estat i pren la sortida oportuna, el codi torna a començar el bucle. Això pot ser un problema en segons quins casos, com el que el sensor a llegir tingui un nombre limita de lectures o calgui un consum del dispositiu molt baix.

R Tot i que els dos exemples que han aparegut sobre FSMs son amb només dos estats, una màquina d'estats en pot tenir un nombre arbitrari segons les necessitats de l'aplicació.

31.3 Codificant FSMs

Com ja hem vist al Llistats 31.3 i 31.6, és relativament senzill codificar una FSM en C. Tot i això, es pot trobar un model genèric per simplificar les coses. Anem a presentar-lo.

Com s'ha dit anteriorment, una FSM consta de 4 operacions:

- Llegir les entrades. Això provocarà o no canvis en l'estat i les sortides.
- Calcular els nous valors de les variables.
- Escriure les sortides que calgui segons l'estat de la màquina d'estats.
- Calcular l'estat següent segons les entrades llegides, les diverses variables i l'estat actual.

Llistat 31.7: Estructura bàsica d'una FSM

```
void loop() {
  read_inputs();
  calc_values();
  write_outputs();
  next_estate();
}

int main(void) {
  ...
  setup();
  ...

  while(1) {
    loop();
  }
}
```

L'ordre d'aquestes operacions és indiferent mentre s'executin totes 4 a cada iteració. D'aquesta manera, podem disposar la nostra funció de *loop()* tal com es veu al Llistat 31.7.

Si adaptem l'exemple del termòstat a aquest mètode de codificació, quedaria tal com es veu al Llistat 31.8 i al repositori.

Si la nostra aplicació requereix més d'una FSM, es poden intercalar les crides a cada operació de manera que es vagin executant alternant les FSMs tal com es veu a la Figura 31.9.

Si s'utilitza habitualment aquest mode de programació, és possible que surti a compte muntar-se una estructura de control pròpia que manegui la crida ordenada d'aquestes funcions i sigui senzill registrar una nova FSM per ser executada concurrentment amb les demès. Un esbòs d'aquest *kernel* podria ser el que es veu al Llistat 31.10.

Aquest *kernel* va executant cada una de les operacions de totes les FSM que s'hi hagin registrat prèviament. D'aquesta manera es pot tenir implementades les FSMs que solucionin la nostra aplicació d'una forma ràpida.

Llistat 31.8: Codi de termòstat amb l'estructura bàsica d'una FSM

```
void read_inputs() {
  temperature = getTemperature();
}

void calc_values() {
  switch(state) {
    case Thermo_OFF:
      termostat_on = false;
      break;
    case Thermo_ON:
      termostat_on = true;
      break;
  }
}

void write_outputs() {
  if (termostat_on) {
    SwitchOn();
  } else {
    SwitchOff();
  }
}

void next_estate() {
  switch(state) {
    case Thermo_OFF:
      if (temperature < MIN_TEMPERATURE) {
        state = Thermo_ON;
      }
      break;
    case Thermo_ON:
      if (temperature > MAX_TEMPERATURE) {
        state = Thermo_OFF;
      }
      break;
  }
}
```

Llistat 31.9: Funció de loop amb múltiples FSMs

```
void loop() {
  read_inputs_FSM1();
  read_inputs_FSM2();
  ...
  read_inputs_FSMN();

  calc_values_FSM1();
  calc_values_FSM2();
  ...
  calc_values_FSMN();

  write_outputs_FSM1();
  write_outputs_FSM2();
  ...
  write_outputs_FSMN();

  next_estate_FSM1();
  next_estate_FSM2();
  ...
  next_estate_FSMN();
}
```

Llistat 31.10: Estructura bàsica d'un *kernel* per múltiples FSMs

```
#define MAX_FSM 10

typedef struct {
  void (*read_input_func)(void);
  void (*calc_values_func)(void);
  void (*write_outputs_func)(void);
  void (*next_estate_func)(void);
} FSM_t;

FSM_t FSM_array[MAX_FSM];  // can manage MAX_FSM FSMs

bool register_FSM(FSM_t &fsm) {
  const int index = 0;
  FSM_array[index] = fsm;
  index++;
}

void loop() {
  int i;

  for(i = 0; i < MAX_FSM; i++) {
    if (FSM_array[i].read_input_func) {
      FSM_array[i].read_input_func();
    }
  }

  for(i = 0; i < MAX_FSM; i++) {
    if (FSM_array[i].calc_values_func) {
      FSM_array[i].calc_values_func();
    }
  }
  ...
}
```

31.4 Flux de dades

TBD

32. Tractament del temps

Tot i que el tractament del temps (deixar passar un cert temps, esperar per un esdeveniment un cert temps, etc.) es pot fer tant amb FSMs com amb un bucle de control senzill, hi ha models pensats que el tracten específicament.

32.1 FSMs amb temps

Una forma molt senzilla d'afegir temps a una FSM és afegir un temps d'espera a la funció *loop()*. Un cop decidit el temps de cada iteració, cal afegir aquest retard (*delay*) al final de cada iteració. Així, el codi de la funció *loop()* quedarà tal com es veu al Llistat 32.1 i al repositori.

D'aquesta manera cada iteració s'executarà amb el període triat sempre que el temps d'execució de les seves operacions no l'excedeixi. Com es veu al codi, el temps a esperar-se de la funció *delay()* es calcula a cada iteració, això permet que cada iteració es faci en el període fixat independentment del temps que hagi transcorregut l'execució de les operacions anteriors. Com que sovint les operacions no transcorren en un temps determinista, inserir un simple *delay(period)* faria que l'execució de cada iteració es fes en un temps diferent.

En aquest model, la funció *delay()* pot manegar opcions de baix consum, de manera que pot fer que el microcontrolador entri a un mode de baix consum mentre transcorre el temps seleccionat.

Llistat 32.1: FSM amb control del temps

```
const int period = 50; // period time in miliseconds

void loop() {
  uint32_t start_time, end_time, iteration_time;

  start_time = RTC_CounterGet(); // time value in miliseconds

  /* FSM */
  read_inputs();
  calc_values();
  write_outputs();
  next_estate();
  /* FSM */

  end_time = RTC_CounterGet();
  iteration_time = end_time - start_time;
  delay(period - iteration_time);
}
```

32.2 Tasques periòdiques

Un altre model força habitual de tractar el temps d'una forma senzilla és fent ús de tasques programades. Aquestes tasques son funcions que es criden de forma periòdica i realitzen les tasques necessàries per l'aplicació final. Així, podem tenir un codi similar al del Llistat 32.2

La funció **Registra_tasca()** registra la funció que se li passa per a que es cridi cada cert temps. El *kernel* s'encarrega de programar algun em timer per a que notifiqui el microcontrolador el temps corresponent i, per exemple, pugui posar en un mode de baix consum el microcontrolador mentre aquest temps no arribi. Com que el microcontrolador es pot despertar per diversos esdeveniments, dins el bucle infinit de la funció **main()** es crida a la funció per a que el *kernel*, si s'escau cridi al a funció registrada, torni a programar el *timer* com calgui i torni a dormir el microcontrolador. El pseudocodi per aquesta funció seria tal com es veu al Llistat 32.3.

32.2.1 Implementació

En el cas de treballar amb la plataforma EFM32 part d'aquestes funcionalitats les tenim implementades a la biblioteca RTCDRV i SLEEP que pertanyen a la biblioteca d'alt nivell EMDRV del fabricant [58].

La primera de les biblioteques, RTCDRV, permet gestionar *timers* virtuals fent servir només un *Timer* real, com el nom indica, es fa servir el *timer* RTC del microcontrolador (veure **Capítol 9 - RTC**) [59]. D'aquesta manera es simplifica tenir múltiples *timers* i permet registrar callbacks per quan un *timer* arriba al seu temps programat de manera que quan un *timer* arriba al seu temps d'expiració es crida a la funció de *callback* registrada.

La segona llibreria, SLEEP, gestionar automàticament els modes de baixa energia del microcontrolador, fent que aquest entri al mode més baix possible segons els perifèrics que es tenen activats. D'aquesta manera, amb una sola crida a una funció de la biblioteca s'aconsegueix posar el microcontrolador en mode de baix consum [60].

D'aquesta manera, la funció *Registra_tasca()* passaria a ser una crida a la funció **RTCDRV _StartTimer()** amb la funció periòdica com a *callback* i la resta de paràmetres com calgui (*rtcdrv-*

Llistat 32.2: Estructura bàsica de les tasques programades

```
void tasca1(void) {
 // codi tasca 1
}

void tasca2(void) {
  // codi tasca 2
}

void main(void) {
  // inicialitzacions
  ...

  // es registren les dues tasques, una es crida cada 5 segons, l'altra cada 15
       segons
  Registra_tasca(tasca1, 5000);
  Registra_tasca(tasca2, 15000);

  while(1) {
    Executa_kernel();
  }
}
```

Llistat 32.3: Estructura bàsica de la funció Executa_kernel()

```
void Executa_kernel(void) {
  Configurar els timers necessaris
  Posar processador en mode de baix consum
  /* Aqui el processador esta suspes esperant algun esdeveniment o que es
      dispari un timer */

  Esbrinar quina tasca toca executar-se
  Executar tasca
  (Opcional) Cridar a una funcio generica de l'usuari
}
```

Llistat 32.4: Estructura bàsica de la funció Executa_tasca()

```
/* Aquesta funcio es crida des d'una ISR, ha de ser curta */
static void TimerCallback(RTCDRV_TimerID_t id, void* param) {
  int timer;

  index = *(int*)param;
  my_timers[index].semaphores = true;
}

void Registra_tasca(mycallback_t func, uint32_t period) {
  static int i = 0;

  my_timers[i].callbacks = func;
  my_timers[i].value = i;

  RTCDRV_AllocateTimer(&my_timers[i].timers_array);
  RTCDRV_StartTimer(my_timers[i].timers_array, rtcdrvTimerTypePeriodic, period,
       TimerCallback, &i);

  i++;
}

static void Executa_tasca(int timeout) {
  int i;

  for (i = 0; i < EMDRV_RTCDRV_NUM_TIMERS; i++) {
    if (my_timers[i].semaphores == true) {
      my_timers[i].semaphores = false;
        if (my_timers[i].callbacks) {
          my_timers[i].callbacks();
        }
     }
  }
}
```

TimerTypePeriodic, el temps en mil·lisegons, etc.). I la funció *Executa_kernel()* no hauria de fer gran cosa.

Aquesta estratègia te un problema, i és que la funció de *callback* se la crida dins del context d'interrupció, cosa no sempre desitjable, ja que les ISR haurien de ser sempre funcions molt curtes, sense gaire funcionalitat, tal com es va explicar a **Capítol 7 - Controlador d'interrupcions**. Per solucionar això es pot canviar una mica l'estratègia i mantenir una estructura que permeti saber quina funció cal cridar i tenir una funció de *callback* única que mantingui aquesta informació. Llavors, a la funció *Executa_kernel()* es comprova si s'ha de cridar alguna funció i llavors es crida des d'allà, fora del context d'interrupció. Això es pot veure al Llistat 32.4.

Aquest model de programació és força senzill i es podria veure com un pas previ a l'ús d'un RTOS on el maneig de les tasques és mes complex i ens ofereixen mecanismes de sincronització i comunicació entre les tasques més enllà de variables compartides.

32.3 Multitasca

Com ja veurem a **Part IV - FreeRTOS**, és possible tenir multitasca en sistemes encastats. Pot ser una bona forma de tenir múltiples tasques funcionant alhora sense haver de gestionar nosaltres

mateixos aquesta complexitat.

Donats l'ús de recursos que es fa i la complexitat a l'hora de programar per aquesta mena de sistemes fa que no sigui la solució idònia per tot tipus d'aplicació encastada. A més, aquesta estratègia pot incloure també FSMs, de manera que una o més tasques de l'aplicació estiguin implementades amb una FSM tal com s'ha comentat a les seccions anteriors.

L'ús de RTOS facilita la comunicació entre tasques,

Aquesta pàgina està en blanc expressament, tot va bé.

Temes avançats

33 Gestió d'excepcions 199
33.1 Exemple detectant errors greus

34 *Shadow Registers* 203

35 Baix cosum 205
35.1 Consideracions prèvies
35.2 Modes d´*sleep*
35.3 Estratègies de baix consum
35.4 *Timers* de baix consum
35.5 Baix consum i RTOS

36 Documentant el codi 211

37 CMSIS 215
37.1 CMSIS-Core
37.2 CMSIS-Driver
37.3 CMSIS-DSP
37.4 CMSIS-RTOS
37.5 CMSIS-DAP
37.6 CMSIS-NN

38 Normes de codificació 217
38.1 *The Power of 10: Rules for Developing Safety-Critical Code*
38.2 MISRA-C
38.3 *Embedded C Coding Standard*
38.4 *JPL Institutional Coding Standard for the C Programming Language*

39 DSP 221

40 C++ vs C 223
40.1 Primer exemple en C++
40.2 Un *driver* en C++
40.3 Conclusions

41 Relació Esquemàtic i FW 231
41.1 Selecció de pin-out
41.2 Selecció de rellotges
41.3 Canvis durant el *layout*
41.4 De la placa de prototipat a PCB pròpia

42 Inicialització del sistema i del llenguatge C 235

43 Treballant amb punt flotant 239

Aquesta pàgina està en blanc expressament, tot va bé.

33. Gestió d'excepcions

Sovint treballant amb sistemes encastats ens trobem amb errors d'origen desconegut que es poden provocar per múltiples causes. Així, per exemple, una divisió per zero, un accés incorrecte a una zona de memòria o un accés a una posició de memòria fora de rang faran que el processador es reiniciï [22, pàgina 102][61, pàgina 318][62].

Aquests casos poden ser molt difícils de trobar si són casos esporàdics, però l'arquitectura ARM té unes característiques que ajuden a detectar-los i trobar-los. En síntesi, el cortex-M llença una interrupció molt prioritària anomenada **HardFault_Handler()** quan succeeix un problema greu del que el processador no pot recupera-se, com una divisió per zero, un accés il·legal a memòria, etc. Abans de cridar a l'excepció, la CPU guarda tot de valors claus a diferents registres, i així per exemple en el registre **PC** s'hi emmagatzema l'adreça de la instrucció executada, així que, en principi, només cal anar a aquella posició de memòria per veure quin ha estat el codi que ha causat el problema. També s'emmagatzema el valor de retorn (la instrucció següent a l'executada que ha causat l'error) al registre **LR** [63].

Així doncs, es pot reescriure la ISR per obtenir les dades que ens informi sobre què ha passat per ajudar-nos a obtenir pistes de quin codi està fallant [64].

33.1 Exemple detectant errors greus

A l'exemple del repositori hi ha un codi que genera diferents errors segons la funció que es cridi i una implementació de **HardFault_Handler()**. Aquesta funció està escrita en assemblador, però el que cal veure és que es crida a la funció **my_HardFault_Handler()** que es qui en realitat fa tota la feina i és la que cal entendre [65].

A la primera part (veure Llistat 33.1) de la ISR es treu per la consola de *debug* la causa de l'excepció (*bus fault*, *memory access*, *divide by zero*, etc.).

Tot seguit es treu per la mateixa consola els valors dels registres que hi ha a l'*stack* per tenir dades que ens permetin localitzar l'error (Llistat 33.2).

Llistat 33.1: Codi HardFault_Handler

```
void my_HardFault_Handler(uint32_t *stack) {
  printf("Error Handler\r\n");
  printf("SCB->HFSR = 0x%08lx\r\n", (uint32_t) SCB->HFSR);

  if ((SCB->HFSR & (1 << 30)) != 0) {
    printf("Forced Hard Fault\r\n");
    printf("SCB->CFSR = 0x%08lx\r\n", SCB->CFSR);

    if ((SCB->CFSR & 0x02000000) != 0) {
      printf("Divide by zero\r\n");
    }
    if ((SCB->CFSR & 0x01000000) != 0) {
      printf("Unaligned\r\n");
    }
    if ((SCB->CFSR & 0x00010000) != 0) {
      printf("Undefined\r\n");
    }
  ...
}
```

Llistat 33.2: Codi HardFault_Handler (continuació)

```
void my_HardFault_Handler(uint32_t *stack) {
  ...
  printf("sp = 0x%08lX\r\n",  (uint32_t) stack);
  printf("r0 = 0x%08lX\r\n",  stack[0]);
  printf("r1 = 0x%08lX\r\n",  stack[1]);
  printf("r2 = 0x%08lX\r\n",  stack[2]);
  printf("r3 = 0x%08lX\r\n",  stack[3]);
  printf("r12 = 0x%08lX\r\n", stack[4]);
  printf("lr = 0x%08lX\r\n",  stack[5]);
  printf("pc = 0x%08lX\r\n",  stack[6]);
  printf("psr = 0x%08lX\r\n"  stack[7]);
  ...
}
```

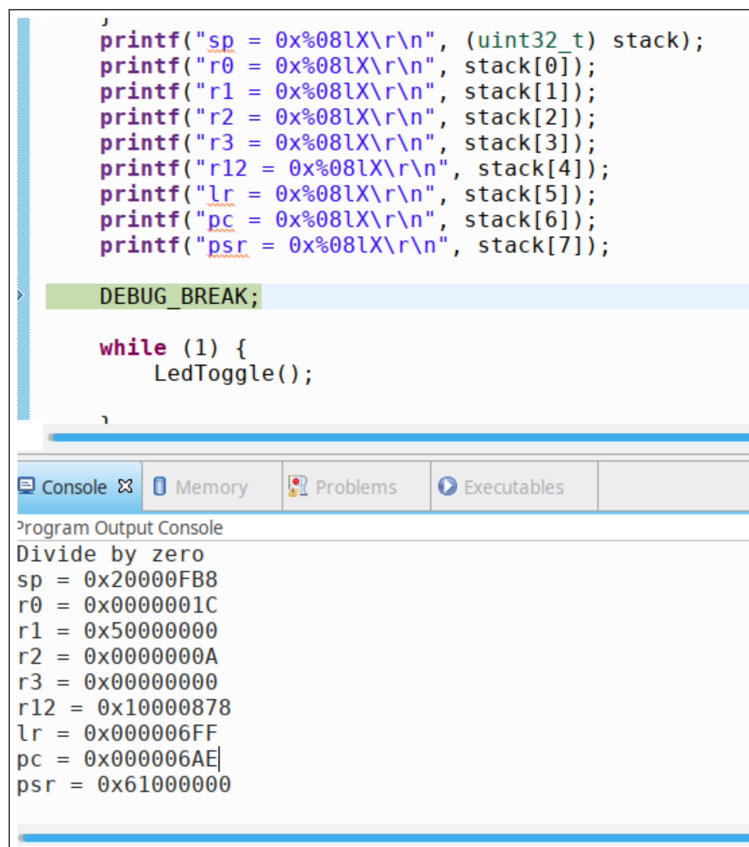


Figura 33.1: Debugger aturat a la instrucció DEBUG_BREAK i el *dump* els registres

```
Disassembly
WrongfunctionAlign
0000069c:   push    {r7}
0000069e:   sub     sp,sp,#0x14
000006a0:   add     r7,sp,#0x0
 21           int a = 10;
000006a2:   movs    r3,#0xa
000006a4:   str     r3,[r7,#0xc]
 22           int b = 0;
000006a6:   movs    r3,#0x0
000006a8:   str     r3,[r7,#0x8]
 24           c = a / b;
000006aa:   ldr     r2,[r7,#0xc]
000006ac:   ldr     r3,[r7,#0x8]
000006ae:   sdiv    r3,r2,r3
000006b2:   str     r3,[r7,#0x4]
 25         }
000006b4:   adds    r7,#0x14
000006b6:   mov     sp,r7
```

Figura 33.2: Codi assemblador a la posició de memòria indicada pel registre **PC**

Per últim, es crida la macro **DEBUG_BREAK**, que està definida com una instrucció en assemblador (**BKPT #01**) que posa el *core* en mode *Debug* i atura l'execució en aquest punt. Així, si tenim un *debugger* connectat, veurem com l'execució s'atura en aquest punt i torna el control a la nostra eina (veure Figura 33.1).

Si anem a la finestra *Disassembly* i anem a la posició de memòria que indica el registre **PC** (0x6AE a l'exemple), veurem que apunta a una instrucció assemblador *sdiv*, que es corresponent amb una divisió. Si mirem el codi anterior, podem deduir que a la posició de memòria **R7+0x8** (corresponent a la variable *b*) s'hi ha emmagatzemat un 0 (instruccions a 0x6A6 i 0x6A8) i aquesta variable es fa servir a la divisió com a divisor, causant l'error (veure Figura 33.2).

També cal comentar que les diferents funcions que generen errors són les següents:

- **WrongfunctionDiv0()** causa una divisió per zero.
- **WrongfunctionAlign()** causa un error d'accés a memòria fora d'alineament.
- **WrongfunctionWrongMemory()** causa un error per accés fora dels límits de la memòria.
- **fp()** causa un intent d'executar a la posició 0x0000_0000 de memòria.

34. *Shadow Registers*

En algunes arquitectures i en perifèrics d'alguns fabricants poden llegir que es fan servir *shadow registers*. S'anomenen així a registres que contenen una còpia d'un altre registre i que son els que es poden llegir per part d'altres dispositius o perifèrics.

Així per exemple, trobem *shadow registers* a alguns processadors de manera que quan la CPU entra a una interrupció es passa a treballar amb un banc separat de registres de propòsit general. Això es fa per evitar un sobrecost a la crida de la ISR, ja que si es tenen aquests registres s'han de guardar els valors actuals de tots els registres a la pila abans de poder executar el codi de la ISR. En canvi, si es tenen aquests registres, la CPU passa a treballar amb un banc diferent (els *shadow registers*) durant l'execució de la ISR i no cal salvaguardar cap valor dels registres originals. Un cop se surt de la ISR la CPU torna a treballar amb el banc de registres originals. En el cas dels Cortex-M no es treballa amb aquesta mena de *shadow registers* i, per tant, caldrà que les ISR salvin els valors dels registres de propòsit general que sobreescriguin durant la seva execució.

Una altra lloc on ens podem trobar *shadow registers* és en alguns perifèrics que treballen valors grans repartits en diversos registres. Si aquests registres s'actualitzessin entremig d'una lectura per part del Firmware, aquest podria tenir una inconsistència a les dades. Per això, és habitual que un valor determinat s'emmagatzemi a *shadow registers* mentre els registres "amagats" s'actualitzen de forma normal. Aquests *shadow registers* seran els que el firmware pot llegir i s'actualitzaran tots de cop una vegada s'hagin llegit tots pel firmware.

R A tots ens ha passat o tenim un company que ha perdut una tarda sencera intentant llegir uns registres d'aquesta mena sense seguir bé l'ordre i rebent valors dolents sense caure en el problema amb els *shadow registers*.

Un exemple d'això últim succeeix amb els registres de data i temps del RTC dels microcontroladors d'ST (veure **Capítol 9 - RTC**). Aquest perifèric conté uns *shadow registers* on es copien cada 2 cicles els registres reals amb la data, el temps i els segons del RTC (Figura 34.1). Quan es llegeix el registre amb el temps o amb els segons es bloqueja la còpia de tots els tres registres perquè la

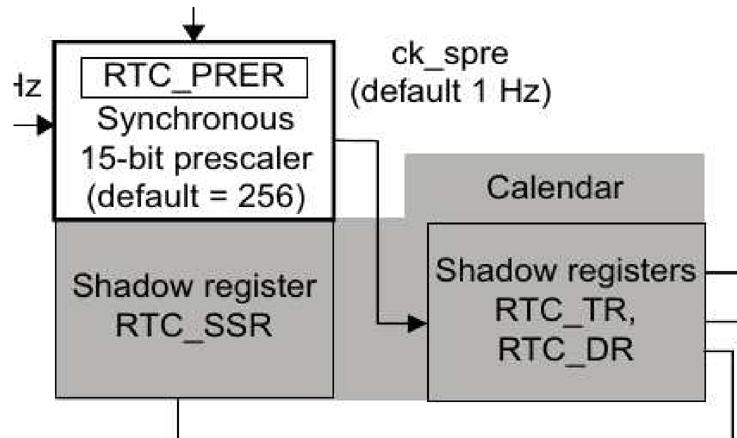


Figura 34.1: *Shadow registers* del perifèric RTC dels STM32 [30, pàgina 800]

lectura dels demès no doni cap incoherència. Si no hi fossin, podria passar que es llegís el temps (per exemple les 23:59:59 del dia 1) i poc després al llegir la data ja hagués passat el segon i la data ja fos el dia 2, resultant en que enlloc de llegir les 23:59:59 del dia 1 s'hauria llegir les 23:59:59 del dia 2. En aquest cas sembla que és molt millor llegir la data correcte i que es tingui un error d'un segon a tenir un dia sencer d'error (!).

Per tant, en aquest perifèric, primer cal llegir el registre amb el temps o els segons i després el registre amb la data [30, pàgines 800 - 805]. Així si només es vol llegir el temps del RTC perquè no interessa la data del sistema, no es poden fer lectures consecutives del temps sense llegir també, encara que no interessi, la data del RTC. La API del fabricant en aquest cas no ho gestiona, però si que ho adverteix a la seva documentació [66, pàgina 719].

35. Baix cosum

Un dels temes més habituals de trobar-se quan es tracten temes amb microcontroladors és el del baix consum. Gràcies a la tecnologia de fabricació dels microxips i els avenços en les arquitectures dels microcontroladors, aquests han arribat a unes fites de consum molt baixes, permeten desenvolupar aplicacions on el sistema pugui anar alimentat per bateries o altres fonts d'alimentació alternatives a l'alimentació general. En aquest capítol veurem les característiques actuals dels microcontroladors en aquest aspecte, com treure tot el partit a aquestes característiques i, per últim, com adaptar els RTOS per treballar amb baix consum.

Cal repassar uns quants conceptes sobre el consum d'energia abans d'introduir-nos de ple en el tema.

35.1 Consideracions prèvies

Per la pròpia natura dels circuits digitals, aquests consumeixen sobretot quan el seu rellotge principal està actiu. Això fa que l'estratègia principal per reduir el consum d'un circuit és desactivar-li precisament el rellotge o reduir la seva freqüència, ja que el consum és proporcional a la velocitat de rellotge.

R Donat que el consum és quasi proporcional a la freqüència de rellotge, els fabricants acostumen a donar el consum per MHz (típicament μA/MHz).

També cal tenir en compte que qui més consumeix en un microcontrolador és el propi *core* o CPU i que, per tant, serà el mòdul que caldrà tenir apagat el màxim de temps possible.

35.2 Modes d'*sleep*

Els diferents fabricants de microcontroladors basats en Cortex-M ofereixen diferents modes d'sleep, això és, diferents combinacions de perifèrics que estan actius a cada mode per tal de reduir el consum.

Taula 35.1: Consum d'energia de diferents fabricants i modes (per un Cortex-M0+) [40][67]

Processador SleepMode	STM32	EFM32
EM0 - *Run mode*	76 μA/Mhz	114 μA/MHz
EM1 - *Sleep mode*	42 μA/MHz	48 μA/MHz
EM4 - *Standby mode*	230 nA	20 nA

Així, els microcontroladors de Silicon Labs tenen 4 modes d'sleep[1] [24, pàgina 6]:

- EM0 - *Energy Mode 0*: Tot el sistema està actiu incloent-hi tots els perifèrics.
- EM1 - *Energy Mode 1*: La CPU està desactiva i la resta de perifèrics estan disponibles.
- EM2 - *Energy Mode 2*: La CPU està desactivada i només els perifèrics de baix consum estan disponibles (UART, RTC, TIMER, Watchdog)
- EM3 - *Energy Mode 3*: Tot el sistema està desactivat, només es manté la RAM activada i certes interrupcions
- EM4 - *Energy Mode 4*: Tot el sistema està desactivat, només es pot fer un *reset* al sistema.

En canvi, els microcontroladors de ST tenen només 3 modes de baix consum[2] [30, pàgina 126]:

- *Run mode*: Tot el sistema està actiu incloent-hi tots els perifèrics.
- *Sleep mode*: La CPU està desactiva i la resta de perifèrics estan disponibles.
- *Stop mode*: Tot el sistema està desactivat, només es manté la RAM activada i certes interrupcions
- *Standby mode*: Tot el sistema està desactivat, només es pot fer un *reset* al sistema.

Els *core* Cortex-M es poden posar en mode de baix consum fent servir dues instruccions **WFI** i **WFE**. El primer que cal fer és configurar a quin mode d'adormir es vol posar el microcontrolador i després executar la instrucció que pertoqui. La CPU es quedarà en l'estat de baix consum que s'hagi configurat fins que es generi una IRQ per algun perifèric o generat per un senyal extern.

35.3 Estratègies de baix consum

Vist tot l'anterior, l'estratègia bàsica per tenir un baix consum serà la de preparar els perifèrics per a que facin la funcionalitat d'entrada/sortida necessària de manera que llencin una IRQ quan finalitzin, posar en un dels modes de baix consum on la CPU està desactivada a l'espera de les interrupcions; a continuació, la CPU processarà les dades o esdeveniments que hagin succeït i es tornarà a configurar els perifèrics i es tornarà a posar la CPU en mode baix consum, etc.

Per tant, quan es desenvolupa una aplicació per ser de baix consum, s'acostuma a treballar basant-se en interrupcions (Veure **Capítol 7 - Controlador d'interrupcions**) i tenint la CPU el màxim de temps en algun dels modes de baix consum.

35.3.1 Exemple de baix consum

L'exemple que es veurà farà servir l'ADC per convertir una entrada analògica a un valor digital, com ja es a fer a l'exemple **Secció 12.1 - Exemple d'ADC**. En el cas de baix consum, es configura el perifèric de la mateixa forma però s'hi afegeix l'opció que generi una IRQ quan acaba de fer una conversió. Així, el nostre codi al bucle principal engegarà la conversió, entrarà en el mode de baix

[1] A més, hi ha el mode normal, on la CPU està a ple rendiment
[2] Versions de Cortex-M0+ tenen algun mode més

Llistat 35.1: Bucle principal amb funcions de baix consum

```
void main() {
  ...
  while (1) {
    ADC_Start(ADC0, adcStartSingle);

    EMU_EnterEM1();

    ADCvalue = ADC_DataSingleGet(ADC0);
    printf("ADC Value %lu\r\n", ADCvalue);
  }
  ...
}
```

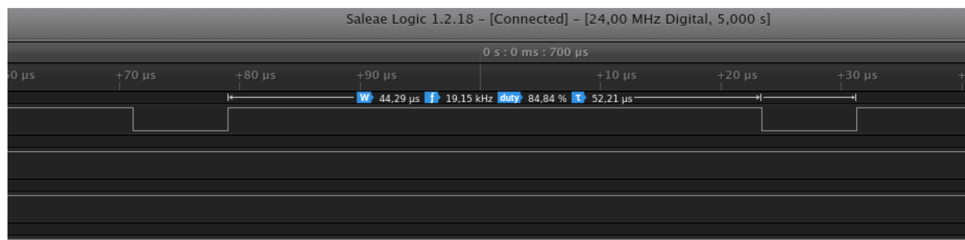


Figura 35.1: Captura de les mesures de temps amb l'analitzador lògic

consum **EM1** perquè la CPU es quedi en repòs mentre l'ADC fa la seva feina i es desperti per la IRQ de finalització; tot seguit es llegeix i es mostra la dada convertida.

Podem fer una mesura del temps que està la CPU en el mode EM1 posant un pin a '1' quan s'entra al mode i posar-lo a '0' quan se'n surt, tal com es veu al projecte d'exemple.

Si usem l'analitzador lògic per mesurar els temps, veiem la imatge de la Figura 35.1 que les mesures diuen que 44,29 microsegons de 52.21 la CPU està en mode de baix consum (el 84.84% del temps).

35.4 *Timers* de baix consum

Un mode que es fa servir sovint en sistemes de baix consum és el de tenir un *timer* configurat perquè desperti el sistema cada cert temps. Així per exemple, en un sistema que ha de llegir un sensor cada 30 segons, el *timer* seria l'únic perifèric en funcionament actiu i estaria configurat per generar una IRQ cada 30 segons; la resta del microcontrolador podria estar en un mode de baix consum que el permeti consumir molt poca energia mentre espera a ser despertat per una IRQ.

Al projecte del repositori hi ha un exemple d'aquest tipus. Es fa servir un **LETIMER**, que és un *timer* de baix consum i baixa freqüència que pot funcionar mentre la resta del microcontrolador està en el mode EM2 (o EM3 segons la configuració que es faci servir) [4, pàgina 294]. Aquest **LETIMER** es pot alimentar amb el rellotge extern de baixa freqüència a 32.768 Hz (**LFXO**) o bé amb l'oscil·lador intern a 1.000 Hz (**ULFRCO**) (al codi es pot triar segons es defineixi o no la macro **USE_ULFRCO**). El rellotge que s'hagi triat es pre-escala per un factor suficient per tenir un comptador prou lent, ja que cal tenir en compte que aquest comptador és de només 16 bits i, per tant, si tenim una freqüència de funcionament elevada no podrem comptar gaire temps. Tot seguit el *timer* es configura per generar una interrupció quan arribi a 0 (és un comptador decreixent) i el seu valor **TOP** (al valor al que es reinicia després d'arribar a 0) es posa en funció de la freqüència

Llistat 35.2: Exemple ús de **LETIMER**

```
#define PRESCALER cmuClkDiv_1
#define EFECTIVE_CLK_FREQ (1000/PRESCALER)
#define SLEEP_SECONDS 4
#define TOP_VALUE (EFECTIVE_CLK_FREQ * SLEEP_SECONDS)

void LETIMER0_IRQHandler(void) {
        uint32_t flags;

        /* Clear flag for LETIMER0 */
        flags = LETIMER_IntGet(LETIMER0);
        LETIMER_IntClear(LETIMER0, flags);

        /* Toggle LED ON/OFF */
        GPIO_PinOutToggle(gpioPortD, 7);
}

void main(void) {
  ...
  /* ULFRCO is 1,000 kHz */
  CMU_ClockSelectSet(cmuClock_LFA, cmuSelect_ULFRCO);
  CMU_ClockDivSet(cmuClock_LETIMER0, PRESCALER);
  ...
  LETIMER_CompareSet(LETIMER0, 0, TOP_VALUE);
  ...
  while (1) {
    /* nothing to do here */
    EMU_EnterEM3(true);
  }
}
```

de funcionament i el temps que es vol tenir el sistema en baix consum, a l'exemple del repositori es posa a 4 segons. Un resum del codi de l'exemple es veu a Llistat 35.2.

Aquest és un exemple senzill que fa servir un *timer* especial de la família EFM32 de Silicon Labs. Altres fabricants proporcionen *timers* similars. Així ST té un *timer* força similar, anomenat LPTIMER ([68]) i Fresscale te el LPTMR amb característiques similars [69]. Els *timers* de ST i de Silicon Labs poden generar senyals tipus PWM mentre el microcontrolador està en modes de baix consum (veure **Capítol 10 - PWM**).

35.5 Baix consum i RTOS

Quan treballem amb un RTOS funcionant en el nostre microcontrolador, hi ha diferents estratègies per aconseguir disminuir el consum energètic.

Bàsicament hi ha dues estratègies:

- Aprofitar la tasca *Idle* per posar al microcontrolador en un mode de baix consum.
- Passar a un sistema sense *tick* (també dit *tickless*).

En qualsevol cas, l'avantatge de que sigui el SO qui s'encarregui de gestionar el baix consum és que les tasques no s'han de preocupar per aquesta gestió.

35.5.1 Tasca *Idle* per baix consum

L'estratègia més senzilla és la d'activar un mode de baix consum quan s'executa la tasca *Idle*. Com que aquesta tasca s'executa quan no hi ha cap altra tasca preparada per agafar el microcontrolador, té sentit pensar en aturar el microcontrolador i esperar a que una tasca estigui disponible. Quan succeeixi el proper *tick*, el microcontrolador sortira del mode d'*sleep* i tornarà a executar el planificador, que, si segueix sense haver cap tasca disponible (en estat *Ready*) per executar tornarà a executar la tasca *Idle* que tornarà a adormir la CPU i es repetirà el cicle [70].

R Cal recordar que quan el *core* està en algun mode de baix consum, el *SysTick* també es desactiva. Per tant, per poder tenir un *tick* quan el *core* està en un mode de baix consum caldrà fer servir un altre Timer que si que funcioni en aquests modes de baix consum.

Cal pensar que tot i que aquest mètode és molt senzill d'implementar, té la limitació de que a cada *tick* es treu la CPU del mode de baix consum per comprovar si hi ha alguna tasca en estat *Ready*. Podem imaginar-nos una aplicació que llegeixi d'un sensor cada 200 ms i processant les dades, com l'aplicació d'exemple XXXXX. Si es té en compte que el *tick* pot ser de 1000 Hz, és fàcil d'observar que es despertarà molts cops al *core* perquè tant sols el planificador vegi que no hi ha cap tasca *Ready* i torni a adormir el processador.

Aquesta característica es pot activar a FreeRTOS editant el fitxer "FreeRTOSConfig.h" i fixant a '0' la definició **configUSE_TICKLESS_IDLE** i triant el valor '1' per **configUSE_SLEEP_MODE_IN_IDLE**. En el cas de Silicon Labs, el microcontrolador es posa en el mode EM2 (veure **Secció 35.2 - Modes d'*sleep***) i deixant en funcionament tant sols el RTC (veure **Capítol 9 - RTC**) i les IRQ dels GPIOs que l'usuari hagi configurat (veure **Capítol 7 - Controlador d'interrupcions**).

35.5.2 FreeRTOS sense *tick*

L'altre estratègia per disminuir encara més el consum, és desactivar el *tick* durant cert temps. En una aplicació on totes les tasques estan bloquejades (i que entraria la tasca *Idle*) es pot calcular el temps en que alguna tasca es desbloquejarà (perquè alguna tasca estigui bloquejada perquè ha cridat la funció vTaskDelay()). Es pot desactivar el *Tick* i programar el Timer perquè generi una interrupció en aquell temps calculat. Si mentre està el sistema adormit esperant aquell temps hi ha algun esdeveniment extern (interrupció), es despertarà i es podrà reprendre l'execució normal i tornar a activar el *Tick*.

Amb aquesta estratègia es maximitza el temps en que el *core* està en algun dels modes de baix consum i per tant es pot reduir dràsticament el consum d'una aplicació (veure **Capítol 35 - Baix cosum**).

En el cas de FreeRTOS, el port disponible per Cortex-M ja incorpora aquesta característica, i es pot configurar editant el fitxer "FreeRTOSConfig.h", concretament fixant el valor '1' a la macro **configUSE_TICKLESS_IDLE**. En el cas de Silicon Labs, el microcontrolador es posa en el mode EM2 igual que en cas amb *ticks* i es programa el RTC perquè generi una IRQ en el temps adequat.

En ambdós casos el codi que gestiona el baix consum i els *ticks* en el port FreeRTOS està al fitxer **low_power_tick_management.c** a la funció **vPortSetupTimerInterrupt()**.

Aquesta pàgina està en blanc expressament, tot va bé.

36. Documentant el codi

Un tema recurrent en temes d'enginyeria del software és com documentar el codi font que es desenvolupa per tal d'afavorir, sobretot, el manteniment del codi durant el temps i algú altre (o nosaltres mateixos) haguem de modificar, re-uilitzar o arreglar algun problema. No farem aquí una discussió sobre els beneficis de documentar, quan fer-ho, etc.

Hi diferents tècniques i mètodes de documentar el codi, aquí veurem només una, basada en Doxygen. Aquest programa processa la documentació inserida dins el propi codi font i genera diferents sortides, la més habitual és una carpeta html amb tota la documentació ben bonica i accessible amb un navegador (té altres formats de sortida, com .pdf, .doc, etc.). Per documentar el nostre codi, el que cal que fem és escriure la documentació dins el propi codi com a comentaris de codi seguint unes normes i *tags* molt senzills propis de Doxygen. (veure Figura 36.1). Aquest mètode de documentar ha esdevingut un estàndard de facto i es troba arreu. Per documentar-se sobre com treballar amb Doxygen, la seva pàgina web està força bé amb exemples de tots tipus [71].

A simplicity (i de fet, a qualsevol IDE basat en Eclipse), podem activar Doxygen com l'eina de documentació, i d'aquesta manera l'editor ens ajudarà alhora d'escriure-la, ja que, per exemple, en escriure «/**» davant una funció ens inserirà automàticament el codi Doxygen per documentar-la (incloent-hi tots els paràmetres), simplificant molt la nostra feina.

Una bona opcio és afegir un directori on ficar-hi el fitxer de configuració del Doxygen (directori /Doc) i on es genera el codi html (directori /Doc/html). El Doxygen s'executa dins del directori /Doc i es genera el codi html (o pdf, o rtf, o el que calgui). Si al fitxer Doxygen li posem l'extensió .doxyfile el propi simplicity el reconeix com a fitxer de documentació i podem executar Doxygen pitjant el botó amb una arroba de color blau a la barra d'eines (Figura 36.2).

També podrem editar de forma visual el fitxer de configuració fent-hi doble-click i veure el resultat obrint dins del Simplicity el fitxer /Doc/html/index.html (Figura 36.3).

Hi ha un exemple complet al projecte FreeRTOS Queue (veure **Subsecció 25.2.1 - Exemple**

```
main.c

    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

/**
 * @brief Our first task. A task is always a endless loop
 *
 * @param pParameter Unused parameter
 * @return none
 *
 */
static void TaskLedToggle(void *pParameter) {

    (void) pParameter;
    uint32_t my_delay;
    uint32_t recv_delay;

    /* Here optional init variables or functions */

    my_delay = pdMS_TO_TICKS(500);
```

Figura 36.1: Comentari per doxygen dins un codi

Figura 36.2: Botons de Simplicity, l'arroba blava permet executar Doxygen

amb cues). En aquest cas, l'explicació del projecte (la secció principal anomenada mainpage en Doxygen) està al final del fitxer main.c. També hi ha la possibilitat de posar aquesta secció en un fitxer a part, normalment un fitxer README.md. Si ho fem així, aquest fitxer README.md github el presenta a la pàgina principal del projecte. El fitxer generat també es pot obrir dins el propi Simplicity Studio (Figura 36.4).

A més, si configurem com cal github, podem pujar el codi html generat per Doxygen al repositori i veure'l a un adreça de github. La de l'exemple està a aquí [72] i es pot obrir des d'un navegador qualsevol,

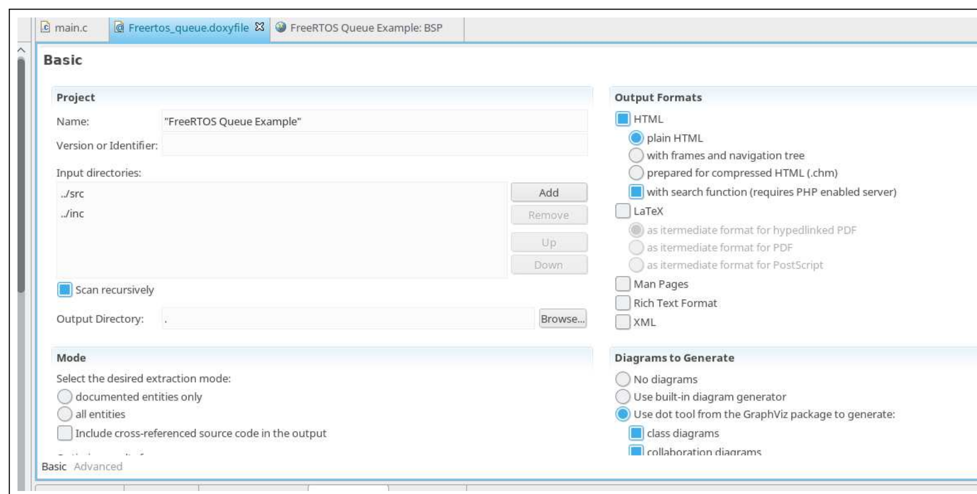


Figura 36.3: Configuració de Doxygen dins de Simplicity

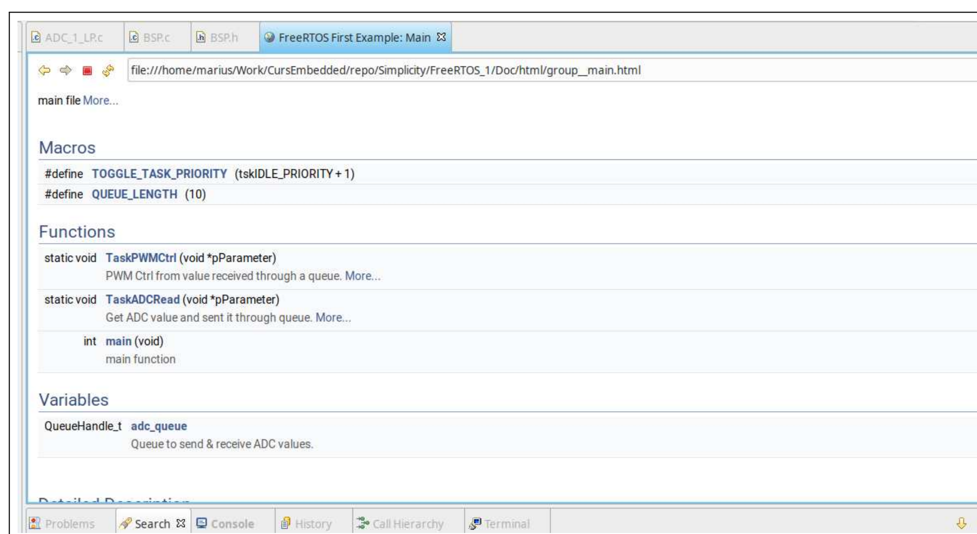


Figura 36.4: Pàgina web de documentació vista dins de Simplicity Studio
Pàgina web de documentació vista dins de Simplicity Studio, en aquest cas es visualitza un dels fitxers locals

Aquesta pàgina està en blanc expressament, tot va bé.

37. CMSIS

CMSIS és una proposta d'ARM per unificar les diferents biblioteques dels fabricants sota una sola especificació, de manera que un disseny es pugui migrar a un altre fabricant de Cortex sense gaires problemes. Hi ha diferents subconjunts d'aquesta proposta, anem a veure'ls un a un.

37.1 CMSIS-Core

Aquesta part de l'especificació fixa la forma de comunicar-se amb les parts més *core* de la CPU, com son: el mapa de memòria (**Subsecció 2.3.1 - Perifèrics mapats a memòria**), el sistema d'excepcions (**Capítol 33 - Gestió d'excepcions**), els registres de control de la CPU, el gestor d'interrupcions (**Capítol 7 - Controlador d'interrupcions**), el Systick (**Secció 5.1 - *Systick***) i les *caches* [73]. En aquesta biblioteca s'inclouen també els fitxers d'inicialització de cada microcontrolador en concret (**Subsecció 2.3.3 - Procés de *boot***).

Així, i a tall d'exemple, les funcions que ja hem fet servir per controlar interrupcions com **NVIC_EnableIRQ()** a **Capítol 7 - Controlador d'interrupcions** no són pròpies de cap fabricant si no que són funcions definides per **CMSIS-core**. També la manera en que es defineixen estructures per accedir als diferents perifèrics ve marcada per l'especificació **CMSIS-Core** (veieu **Llistat 2.2 - Exemple de definició d'estructura per accedir a memòria**).

37.2 CMSIS-Driver

Aquesta especificació defineix una API[1] per tot un seguit de perifèrics per tal que els fabricants puguin implementar el *driver* corresponent i els desenvolupadors no hagin de dependre de llibreries pròpies de cada fabricant. Aquesta especificació inclou els següents perifèrics:

- CAN
- Ethernet
- I2C

[1] *Application programming interface*

- MCI[2]
- NAND
- Flash
- SAI[3]
- SPI
- Storage
- USART
- USB

Hi ha una implementació d'alguns dels mòduls feta per l'autor al repositori `https://github.com/mariusmm/CMSIS_Drivers` pels dispositius de Silicon Labs i de ST.

37.3 CMSIS-DSP

Aquesta biblioteca inclou totes les funcions específiques de tipus DSP dels Cortex-M més avançats (Cortex-M4 i Cortex-M7) i funcions que treballen amb punt flotant per tot tipus de Cortex-M. Si el Cortex-M amb el que treballem suporta punt flotant, la biblioteca farà les operacions per HW, i les farà per SW en cas contrari [74][75].

37.4 CMSIS-RTOS

Aquesta biblioteca defineix un conjunt de funcions i crides per "amagar" el sistema operatiu que es pugui fer servir, de manera que es pugui intercanviar el RTOS sense afectar al codi d'aplicació [76].

D'aquesta manera es tenen crides estàndard per les funcions habituals (crear tasques, semàfors, cues, etc., enviar dades a la cua, etc.) i així es pot, en principi, intercanviar el RTOS sense haver de canviar res del codi d'usuari. Fent servir aquesta API no cal conèixer les interioritats i particularitats de cada RTOS que es vulgui fer servir, ja que quedaran amagades i pre-configurades per la biblioteca.

Així tenim que ST proporciona un *wrapper* de CMSIS-RTOS per FreeRTOS que s'integra fàcilment al seu IDE [77]. Silicon Labs no proporciona suport per aquesta biblioteca, però es pot fer servir el *wrapper* de codi obert disponible a GitHub.

A part, es va crear una implementació de CMSIS-RTOS anomenada CMSIS-RTOS-RTX (o també Keil RTX) per part de Keil (empresa propietat d'ARM) [78].

37.5 CMSIS-DAP

Més que una biblioteca, aquesta part de CMSIS és una definició de com ha de treballar un dispositiu que faci de pont entre un port USB i el port de configuració dels microcontroladors Cortex. Això possibilita que, per exemple, la placa de prototipat tingui un port USB i el puguem fer servir per programar el microcontrolador, tenir la consola de *debug* (SWO), poder inspeccionar registres de la CPU, etc. [79].

37.6 CMSIS-NN

Aquesta biblioteca està composta d'un seguit de funcions i algorismes per implementar xarxes neurals a processadors Cortex-M i queda fora de l'objectiu d'aquest llibre [80][81].

[2] *Memory Card Interface*
[3] *Serial Audio Interface*

38. Normes de codificació

Per tal d'unificar estils de codi i per evitar possibles errors, és habitual seguir algun conjunt de normes de codificació quan es desenvolupa un projecte. Aquest costum de normes acostumen a ser una llista de recomanacions d'estil sobre l'escriptura del codi, normes sobre coses prohibides o no recomanades, etc, Per cara regla, s'acostuma a donar una breu explicació del motiu. Aquests conjunts de normes acostumen a ajudar a evitar *bugs* de difícil detecció.

R Tot i que un conjunt de normes de codificació ajuda a no inserir *bugs*, les normes per si soles no poden garantir que no es generin *bugs* en un sistema complex. Cal sempre seguir les bones pràctiques de Test.

Normes generals n'hi ha moltes i alguna de les mes populars és la coneguda com "The Power of 10: Rules for Developing Safety-Critical Code" ("El poder del 10: regles per desenvolupar codi crític" [82]. En aquest document es presenten tant sols només 10 regles per ajudar a escriure codi més segur i menys propens a errors.

En àmbits molt específics hi ha normes i estàndards propis, com el DO-178 per l'àmbit aeri i espacial; IEC 61508, ISO 26262 o SAE J3061 per automoció o IEC 62304 per l'industria mèdica. Per l'àmbit espacial el JPL (*Jet Propulsion Laboratory*) té publicada una norma pròpia [83].

També hi ha normes genèriques, que no es centren a cap àmbit concret. Les normes genèriques més habituals i conegudes són MISRA-C [84] i "Embedded C Coding Standard" [85]. Per espai, la ESA[1] fa servir el document "C and C++ Coding Standards" [86].

Es descriuen breument als apartats següents.

[1] *European Space Agency*

38.1 *The Power of 10: Rules for Developing Safety-Critical Code*

Aquest conjunt de només 10 regles es va escriure per ajudar a l'anàlisi estàtic del codi i la revisió per desenvolupadors. Es poden resumir en:

- Evitar construccions complexes com *goto* i l'ús de recursivitat.
- Tots els bucles han de tenir fitada la seva longitud.
- Evitar l'ús de memòria dinàmica.
- Restringir la llargada d'una funció a 60 línies.
- Fer servir un mínim de dos comprovacions en temps d'execució per cada funció.
- Restringir la vida de les dades el més possible.
- Comprovar el valor de retorn de totes les funcions que retornen un valor.
- Poc ús del pre-processador.
- Limitar l'ús de punters a una sola indirecció i no usar punters a funcions.
- Compilar amb tots els *warnings* activats. Resoldre sempre tots els *warnings* abans de publicar el codi.

38.2 MISRA-C

MISRA C és un conjunt de normes i guies per programar en codi C per sistemes encastats. Es va proposar per primer cop el 1997 per l'associació MISRA (sigles de *Motor Industry Software Reliability Association*) i ha tingut diverses revisions, la tercera i última es va publicar el 2012 [84][87]. Aquestes especificacions cal comprar-les (la versió digital costa 15 lliures) i no es poden redistribuir lliurement, però si podem tenir accés a algun addenda per veure com són aquestes normes [88].

Aquestes normes es divideixen en 3 classificacions segons el grau d'obligatorietat:

- *Mandatory* són normes que s'han de complir sense cap excepció
- *Required* són normes a complir però es poden incomplir si hi ha una explicació racional (anomenada *Deviations*
- *Advisory* que són normes optatives, però no cal complir-les, tot i que es recomana fer-ho.

Les normes consten d'una frase dient què s'ha de fer o no s'ha de fer, una explicació del perquè de la norma i un exemple de l'ús correcte.

Així si mirem a l'addenda 1 [88, pàgina 4] (que és de lliure distribució i accés), la regla 21.14 diu que la funció **memcmp()** no s'ha de fer servir en altre cosa que no siguin cadenes acabades en NULL ('\0'). Aquesta norma evita que es puguin fer servir *buffers* d'una mida superior a la cadena de text que guarden i provoqui errors que poden ser molt complexes de trobar.

Existixen eines que automàticament comproven la conformitat d'un projecte o codi a les normes MISRA. Entre aquestes eines, algun compilador fa la comprovació en temps de compilació (ho fan els compiladors d'IAR i de TI).

Per últim, cal dir que hi ha força controvèrsia amb d'idoneïtat de seguir les normes MISRA, donat les limitacions que provoca al desenvolupador i les suposades avantatges que proporciona.

38.3 *Embedded C Coding Standard*

Aquestes normes són de lliure accés i escrites pel Barr Group. Conté regles tant d'estil de text (número de caràcters per línia, on posar els '{', etc.) com regles de sintaxi en C, com per exemple quan i on usar la paraula reservada *volatile*, etc. Segons el mateix document, aquestes regles són més laxes que les normes MISRA [85].

En aquest cas, cada regla consta de l'explicació de la regla en si mateixa, el raonament que hi ha per definir la regla, quan pot haver-hi una excepció i com aplicar-la.

També hi ha eines per comprovar que el codi escrit segueix aquestes normes.

38.4 JPL Institutional Coding Standard for the C Programming Language

Aquesta normes de codificació venen d'un laboratori del JPL per tal d'aconseguir millor seguretat i qualitat en el software que s'escriu a les sondes espacials d'aquesta institució [89].

Les normes de codificació son una ampliació de les normes MISRA per afegir-hi sistemes multitasca [83]. Es defineixen nivells d'acompliment amb les normes, anant des de LOC-1 fins a LOC-4 amb un total de 120 regles. La majoria de regles son equivalents a algunes de les normes MISRA. Els dos últims nivells d'acompliment (LOC-5 i LOC-6) consisteixen a acomplir amb totes les regles obligatòries o opcionals de les normes MISRA.

Així, com a diferència de regles que es poden trobar a d'altres normes de codificació, aquestes afegeixen regles com la Regla 6, que demana que sempre es facin servir mecanismes IPC per comunicar tasques entre si, i que cap tasca ha d'accedir a dades o executar codi d'altres tasques. La Regla 7 demana que les tasques no se sincronitzin fent servir *delay*.

Aquesta pàgina està en blanc expressament, tot va bé.

39. DSP

Com ja s'ha comentat, els Cortex-M4 i Cortex-M7 suporten instruccions addicionals de tipus DSP [23, pàgina 173][22, pàgina 255]:

- Instruccions tipus SIMD[1]
- Instruccions de saturació
- Instruccions addicionals de multiplicació i MAC[2]
- Instruccions de empaquetar i desempaquetar
- Opcionalment, instruccions de punt flotant

Aquestes instruccions s'afegeixen al conjunt d'instruccions màquina de la CPU i permeten que els processadors Cortex-M puguin implementar algorismes de DSP de forma prou eficient. Com que moltes d'aquestes instruccions i nous tipus de dades no són estàndard dins els compiladors de C més habituals, ARM proporciona la biblioteca CMSIS-DSP (veure **Secció 37.3 - CMSIS-DSP**). Aquesta biblioteca, curiosament, es pot fer servir tant en Cortex-M4 i M7, com en Cortex-M3 i M0 que no tenen instruccions específiques de DSP.

Per fer-la servir cal fer, almenys, dues passes:

1. Definir un símbol de compilació segons el processador amb el que estiguem treballant (**ARM_MATH_CM0**, **ARM_MATH_CM3**, **ARM_MATH_CM4**).
2. Afegir la biblioteca pre-compilada al nostre projecte (er això cal afegir també el **PATH** on està situada la biblioteca) tal com es veu a la Figura 39.1.

La documentació de la biblioteca proporciona totes les funcions implementades així com un conjunt d'exemples dels usos més comuns [**CORE-DSP**]. SiliconLabs també proporciona documentació en un *Application Note* sobre la biblioteca [75].

[1] *Single Instruction, Multiple Data* Única instrucció, Múltiples dades
[2] *Multiply and Accumulate* Instrucció de multiplicar i acumular

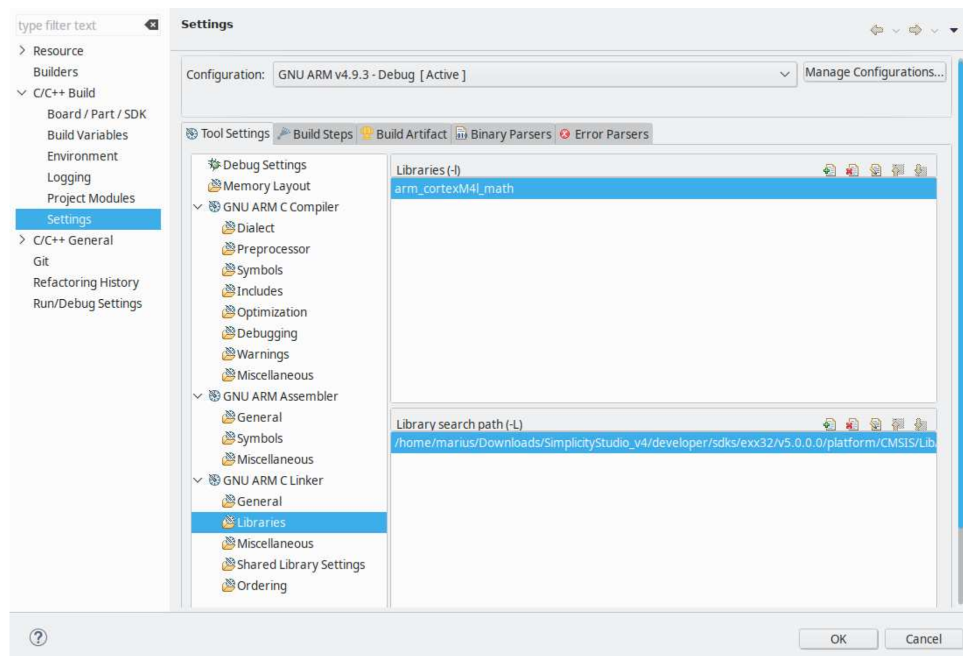
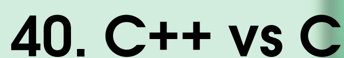


Figura 39.1: Configuració del Simplicity Studio afegint-hi la biblioteca CMSIS-DSP



40. C++ vs C

En aquest llibre s'ha treballat exclusivament en llenguatge C (versió C99) i no s'ha parlat res de C++. Anem a fer-ho ara en aquest capítol.

La discussió sobre usar o no C++ en sistemes encastats deu ser tant antiga com l'aparició d'aquest llenguatge orientat a objectes. Si bé als seus inicis el llenguatge presentava força problemes, ja fa molts anys que és un llenguatge estable i candidat a ser usat en sistemes encastats. Tot i això, la seva popularitat ha estat desigual i encara hi ha molts equips de desenvolupadors de sistemes encastats que treballen exclusivament en C.

Els problemes habituals que s'ha acusat al C++ per no fer-lo servir en sistemes encastats són els següents [90]:

- codi més llarg: si bé això pot ser veritat, les mides de les memòries FLASH dels microcontroladors és cada cop més gran i els compiladors moderns generen codi força optimitzat, a més que es poden desactivar opcions del llenguatge que no es fan servir.
- més lent: això era cert amb els primers compiladors de C++, però actualment el codi generat és de la mateixa qualitat que el generat pels compiladors de C.
- més *stack*: seguint les mateixes normes que amb C, és possible tenir codi C++ que faci un ús correcte de l'*stack*

En canvi, els avantatges que ens pot proporcionar treballar amb C++ poden ser:

- comprovació de tipus en temps de compilació. C és força laxe en aquest tema, i això pot conduir a errors. C++ és capaç de fer comprovacions en temps de compilació per avaluar la correcció de les conversions.
- *namespaces*, que permeten classificar i organitzar el codi d'una forma intuïtiva i senzilla.
- constructors i destructors permeten inicialitzar i destruir o netejar estructures de forma automàtica.
- orientació a objectes, l'organització del codi en objectes pot ajudar a ordenar i simplificar el codi.

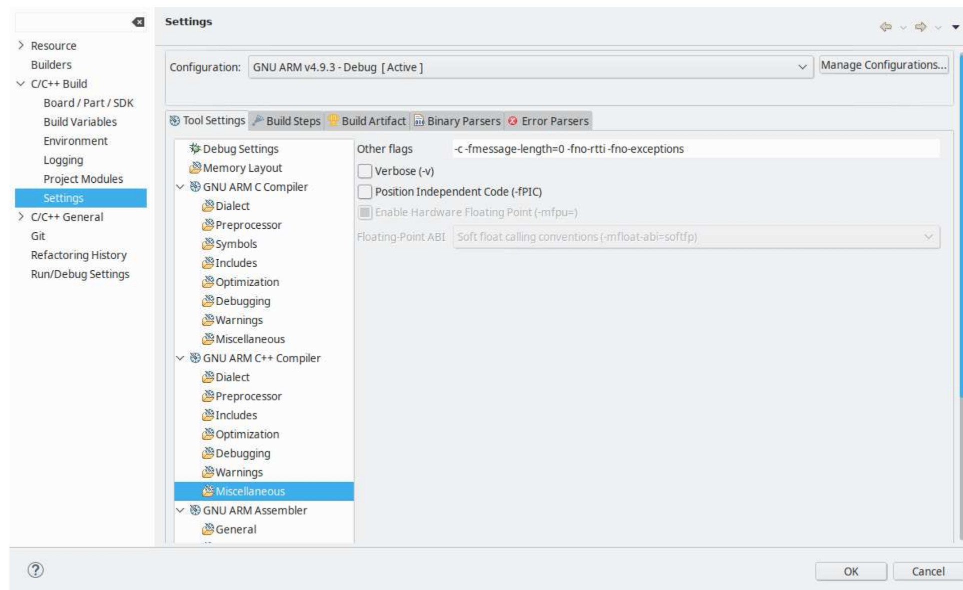


Figura 40.1: Configuració Simplicity Studio per deshabilitar RTTI i les excepcions

- sobrecàrrega d'operadors, fent que operacions entre objectes sigui senzilla amb un codi resultant força senzill.

També cal recordar que no cal fer servir totes les noves capacitats de C++ respecte a C de cop, si no que es poden anar incorporant poc a poc al nostre codi conforme anem guanyant experiència i coneixements.

Dues de les característiques de C++ que ocupen força memòria són el RTTI[1] i el control d'excepcions. RTTI dona informació del tipus de classes polimòrfiques (que tenen almenys un mètode virtual) i és una característica que es faci servir gaire en sistemes encastats. El control d'excepcions permet l'execució d'un mètode i capturar l'error que es pugui generar i tractar-lo fora de la funció i de forma controlada.

Aquestes dues característiques de C++ afegeixen força codi a qualsevol projecte amb el que treballem, fent que, per exemple, no puguem compilar un simple "Hello World embedded" per la nostra placa de desenvolupament ja que ocupa massa FLASH. Les opcions per deshabilitar aquestes funcions al compilador GNU (que és el compilador utilitza Simplicity Studio) son:

```
-fno-rtti -fno-exceptions
```

i es configura tal com es veu a la Figura 40.1.

40.1 Primer exemple en C++

L'exemple CXX_1 és el típic "Hello World" per sistemes encastats escrit en C++.

Aquest exemple fa servir dues classes dins el *namespace* **BSP**.

40.1.1 LED

Com el seu nom indica, serveix per controlar l'únic LED de la PCB de prototipat. Està basada en una classe amb tres mètodes senzills per controlar un sol LED (LED::On(), LED::Off(), LED::Toggle()).

[1] *Run-time type information* Informació de tipus en temps d'execució

Llistat 40.1: Part del codi de la classe LED

```
LED::LED() {
  CMU_ClockEnable(cmuClock_GPIO, true);
  GPIO_PinModeSet(gpioPortD, 7, gpioModePushPullDrive, 0); /* LED */
}
...
void LED::On() {
  GPIO_PinOutSet(gpioPortD, 7);
}
```

Dins el constructor s'activa el rellotge pel perifèric GPIO i es configura el pin corresponent al LED de la PCB (Llistat 40.1).

40.1.2 Button

Aquesta classe gestiona el valor d'una entrada del GPIO d'una fora senzilla, la classe **Button** emmagatzema els paràmetres d'un pin d'E/S i abstreu les crides a la biblioteca **emlib** de Silicon Labs (veure Llistat 40.2).

40.1.3 Un *Hello World* "més C++"

A continuació modifiquem l'exemple per donar-li una volta més i que sigui més "estil C++" (està al repositori). El que s'ha fet ha estat crear una nova classe **Pin** que abstrau la informació d'un pin GPIO d'EFM32. La classe **Button** fa servir **Pin** per obtenir les característiques del GPIO a controlar.

40.1.4 Mida dels executables

A l'exemple CXX_1 tenim el "Hello World embedded" fet en C++ de manera bàsica. A l'exemple CXX_2 s'ha fet una implementació "més C++" amb la mateixa funcionalitat. A la Taula 40.1 es pot veure la quantitat de memòria de tot tipus que necessiten les dues aplicacions així com l'exemple bàsic en C.

Taula 40.1: Ocupació de memòria de "Hello World embedded " en C i C++ (tots els projectes compilats amb optimització -O2).

Aplicació	text	data	bss
GPIO_1	972	108	28
CXX_1	1836	112	32
CXX_2	2076	112	32

Com a curiositat, l'ús de *std::cout* de la biblioteca *iostream* i l'operador **<<** afegeix uns 150KB de codi FLASH (!!!), fent que sigui poc recomanable o impossible de fer servir en un sistema encastat actual.

Llistat 40.2: Part del codi de la classe LED

```
Button::Button(GPIO_Port_TypeDef port, int pin, bool pull, bool pullup) {

  CMU_ClockEnable(cmuClock_GPIO, true);

  m_port = port;
  m_pin = pin;
  m_pull = pull;
  m_pullup = pullup;

  if (m_pull == false) {
    GPIO_PinModeSet(port, pin, gpioModeInput, 0);
  } else {
    if (m_pullup == true) {
      GPIO_PinModeSet(port, pin, gpioModeInputPull, 1);
    } else {
      GPIO_PinModeSet(port, pin, gpioModeInputPull, 0);
    }
  }
}

bool Button::getValue() {
  unsigned int pin_value;

  pin_value = GPIO_PinInGet(m_port, m_pin);
  if (pin_value == 0) {
    return false;
  } else {
    return true;
  }
}
```

40.2 Un *driver* en C++

Com hem vist al llarg del llibre, bona part del codi són *drivers* per controlar els diferents perifèrics o dispositius del nostre sistema encastat. Si treballem en C++, caldrà que aquest *drivers* els fem també en C++. Veurem ara un exemple amb la UART, escrivint un *driver* i un exemple igual al vist a **Secció 14.3 - Un exemple amb la UART més complicat**.

En aquest exemple tenim una classe UART que és la implementació del *driver* per la UART que es va veure a l'exemple de la Secció 14.3. Aquesta classe **UART** fa servir *buffers* circulars per emmagatzemar les dades que es reben o s'han d'enviar per la UART i té els mètodes **AvailableData()**, **GetData()** i **SendData()** com ja tenia el mòdul UART de l'exemple en C. Aquests mètodes tant sols accedeixen al *buffer* circular adequat (de transmissió o recepció) que està implementat a la classe **CircularBuffer** .

Tal com es veu al Llistat 40.4 s'ha sobrecarregat l'operador **<<** per fer més fàcil l'ús de la classe a l'hora d'enviar dades i poder escriure codi com el del Llistat 40.3.

La resta del codi és prou autoexplicatiu a excepció de l'implementació de les ISRs de la UART. En aquest cas ens trobem que les ISRs haurien d'estar encapsulades dins la pròpia classe UART però això no és possible, donat que la classe no és estàtica, i per tant "no existeix" fins que no es crea instanciant un objecte d'aquest tipus [91][92]. Una possible solució a aquest problema és el que es veu al codi 40.5: es té el codi pròpiament dit de la ISR a uns mètodes privats de la classe del *driver* (en aquest cas la classe UART) i en algun altre lloc del codi (en aquest exemple al fitxer *main*) s'insereix la construcció que es veu al Llistat 40.6. D'aquesta manera les ISRs criden als mètodes adequats de la classe pertinent.

40.2.1 Ocupació de memòria

De nou, anem a analitzar l'espai de memòria necessari per aquest exemple comparat amb l'exemple escrit en C amb la mateixa funcionalitat.

El codi en C++ es compila amb 3 variants:

- Sobrecarregant l'operador **<<** que pugui rebre dades de tipus *char*.
- Sobrecarregant l'operador **<<** que pugui rebre dades de tipus *std::string*.
- Sense sobrecarregar l'operador.

Els resultats es mostren a la Taula 40.2. Es pot veure que l'ús de l'operador que suporta *std::string* afegeix força codi ROM (segona columna a la Taula, uns 2 KB) i que, en general, l'ús de C++ afegeix un sobrecost en espai ROM al nostre codi. Potser el més destacable és que la quantitat de RAM necessària no s'incrementa de manera significativa, sent aquest recurs el més escàs en un microcontrolador.

Llistat 40.3: Ús de l'operador << de la classe UART

```
my_uart << "Testing" << " C++ string style";
```

Llistat 40.4: Implementació de l'operador << per la classe UART

```
class UART {
  ...
  UART& operator<<(char* str) {
    for(char* it = str; *it; ++it) {
      this->Tx(*it);
    }
    return *this;
  }

  UART& operator<<(std::string str) {
    for(std::string::iterator it = str.begin(); it != str.end(); ++it) {
      this->Tx(*it);
    }
    return *this;
  }
  ...

  void UART::Tx(unsigned char c) const {
    USART_Tx(m_uart, c);
  }
  ...
}
```

Llistat 40.5: Implementació de les ISRs en C++

```
void UART::USART1_TX_IRQHandler(void) {
  USART_IntClear( USART1, USART_IEN_TXC);
  Send();
}

void UART::USART1_RX_IRQHandler(void) {
  char data;

  if (USART1->IF & LEUART_IF_RXDATAV) {
    data = USART_Rx(USART1);
    m_RX.PushData(data);
    USART_IntClear( USART1, USART_IEN_RXDATAV);
  }
}

class UART {
  ...
  friend void USART1_TX_IRQHandler();
  friend void USART1_RX_IRQHandler();

private:
  void USART1_TX_IRQHandler(void);
  void USART1_RX_IRQHandler(void);
  ...
}
```

Llistat 40.6: Part del fitxer UART.cpp de l'exemple d'us del *driver* en C++ per la UART

```
static UART* helper_uart;

void USART1_TX_IRQHandler() {
  helper_uart->USART1_TX_IRQHandler();
}

void USART1_RX_IRQHandler() {
  helper_uart->USART1_RX_IRQHandler();
}
```

Taula 40.2: Ocupació de memòria d'exemple amb la UART en C i C++ (tots els projectes compilats amb optimització -O2, en KB).

Aplicació	text	data	bss
Sense operador <<	4636	120	40
Amb operador << i char	4644	120	40
Amb operador << i std::string	6796	128	168
Original en C	2620	116	184

40.3 Conclusions

Tot i que l'ús de C++ enlloc de C incrementa la mida de l'executable final i les seves necessitats de memòria, el seu ús pot estar justificat en casos on l'encapsulació que proporciona C++ ajudi a la claredat del codi o a la portabilitat del mateix a diferents plataformes.

En qualsevol cas, cal una expertesa en el llenguatge per fer-ne un bon ús per tenir en compte les particularitats d'escriure codi C++ per sistemes encastats.

Aquesta pàgina està en blanc expressament, tot va bé.

41. Relació Esquemàtic i FW

Quan es dissenya un sistema encastat, una de les parts més importants i on contribueixen perfils professionals de diferent mena és la del disseny de l'esquemàtic. Aquest document especifica tots els detalls hardware de la connexió dels dispositius del sistema, els diferents dominis d'alimentació, els diferents rellotges del sistema, etc. Tots aquests aspectes influiran i seran influïts per, entre d'altres, el disseny de FW i les característiques particulars del microcontrolador triat. És per tot això que en aquesta primera fase de disseny, cal implicació de part de l'equip de FW.

41.1 Selecció de pin-out

Els microcontroladors actuals tenen la capacitat de poder cablejar la sortida o entrada d'un dels perifèrics a diferents pins del mateix. Per exemple, els dos pins del bus I2C (SCL i SDA, veure **Capítol 15 - I2C**), en cert dispositiu de Silicon Labs (EFM32TG840) es poden cablejar cap a: PA0/PA1, PD6/PD7, PC6/PC7, PF0/PF1, PE12/PE13 [31, pàgina 50]. A nivell de FW serà indiferent fer servir un o altre conjunt de pins (tant sols caldrà canviar la configuració) però a nivell d'esquemàtic i a l'hora de fer la PCB pot ser un canvi important.

Un altre aspecte a tenir en compte serà el dels pins d'entrada que poden o no generar IRQ, de quina mena, etc. Per exemple, ja s'ha comentat que a la família STM32 de ST, els pins amb el mateix nombre generen la mateixa interrupció, així el pin PE6 genera la mateixa interrupció (EXTI6) que el pin PA6 [30, pàgina 382]. De forma similar, a EFM32 els pins generen una interrupció o una altra segons tinguin numeració parell o senar i només un pin de cada conjunt amb el mateix nombre pot generar interrupció (només un dels pins de cada conjunt pot generar IRQ: PA1, PB1, PC1, PD1, etc.) [4, pàgina 471]. Aquestes particularitats de cada família poden ser un inconvenient pel disseny FW del sistema i caldrà tenir-ho en compte alhora de dissenyar l'esquemàtic i el FW associat.

41.2 Selecció de rellotges

Un altre aspecte important és el de triar la freqüència de funcionament del rellotge del sistema i d'altres rellotges auxiliars. Com ja s'ha comentat a **Capítol 35 - Baix cosum**, la freqüència de

funcionament del sistema és un dels factors més importants en el consum del microcontrolador. Com és evident, també afecta de forma directa al rendiment del sistema i a la seva capacitat de càlcul, procés de dades i resposta a esdeveniments.

També, però, és important la freqüència triada per la generació d'altres freqüències que necessitin alguns perifèrics. Per exemple, la USART necessita certes freqüències de rellotge per poder treballar amb els *bit-rates* més habituals. Els fabricants proporcionen mètodes per calcular les millors opcions de freqüències segons el *bit-rate* desitjat o taules amb paràmetres precalculats [4, pàgina 153] [30, pàgina 980].

Cal tenir en compte que, sovint, els microcontroladors tenen més d'un arbre de rellotges (veure **Capítol 5 - Gestió de rellotges**) i que cal triar bé quins oscil·ladors i a quina freqüència treballaran.

41.3 Canvis durant el *layout*

Per últim, succeeix sovint que certes connexions de l'esquemàtic es canvien en l'etapa de *layout* per necessitats del disseny. Pot ser que per poder *routejar* millor una línia es demani de canviar de pin. Això provocarà canvis en el FW que s'hauran de tenir en compte. Si hem fet bé el disseny del nostre codi, només caldrà fer algun canvi senzill al nostre BSP (veure **Secció 6.2 - BSP**).

41.4 De la placa de prototipat a PCB pròpia

Un altre dels canvis importants és el de passar de treballar amb una placa de prototipat o de desenvolupament a poder-ho fer en una PCB pròpia. La placa de prototipat porta muntat cert nombre de dispositius externs que segurament no estaran presents a la nostra PCB.

41.4.1 Mecanisme de programació

Un dels canvis més notoris és de l'absència del programador integrat a la PCB pròpia. Les plaques de desenvolupament modernes acostumen a integrar el programador, de manera que la placa de prototipat s'alimenta i es programa a través d'un connector USB estàndard. Això amaga que a la pròpia placa de desenvolupament hi ha tota el circuit per programar el microcontrolador principal. De fet, a la placa de prototipat que estem fent servir, hi ha un microcontrolador que rep les comandes del *debugger* per USB i les transforma a les comandes adequades per programar el microcontrolador principal a través del port *SWD* [93, pàgina 30].

Aquest circuit no s'acostuma a posar les PCBs de productes finals, si no que es deixa disponible d'alguna manera (connector, pins, *pads*) a la PCB l'accés directe al port *SWD* del microcontrolador. Això provoca que calgui un programador extern a la PCB per tal de poder programar el microcontrolador. Això es pot fer comprant un dispositiu *debugger*, tot i que també es pot fer servir una de les plaques de prototipat perquè faci de programador de qualsevol microcontrolador extern connectat a través del port de DEBUG.

A part d'aquesta mena de programació pel port SWD, es pot tenir en compte que alguns microcontroladors porten un *bootloader* en HW que permet actualitzar el Firmware via un port sèrie (veure **Secció 18.2 - *Bootloaders***) [44]. Això pot simplificar la programació del microcontrolador, però cal tenir en compte que aquesta mena de comunicació no permet *debug*.

41.4.2 Migració vertical

Treballant amb la placa de prototipat es poden avaluar les prestacions del sistema així com les necessitats de memòria, tant FLASH com RAM. Usualment, i per temes de costos, s'acostuma a triar el microcontrolador de la mateixa família més senzill i barat que compleixi els requeriments trobats.

Aquest canvi també pot causar canvis al nostre FW, ja que pot ser que diferents models tinguin un mapat de pins diferents, o algun perifèric no es pugui *routejar* al pin que s'havia previst, etc. Aquesta informació acostuma a estar al *Reference Manual* de cada família, a [4, pàgina 8] es veu la taula resum de cada model (anomenats *parts* en anglès).

Per migració vertical ens refereix a la possibilitat de canviar de família dins un mateix fabricant mantenint el mateix encapsulat físic de manera que es poden incrementar les capacitats del microcontrolador sense haver de fer canvis a la PCB. Per exemple, es pot consultar els *datasheets* de la família *Tiny Gecko* (Cortex-M3) [94, pàgina 72], *Zero Gecko* (cortex-M0+) [95, pàgina 66] i *Happy Gecko* (Cortex-M0+)[96, pàgina 76] i es veurà que el mateix encapsulat, per exemple un QFN24, és compatible pin a pin amb qualsevol de les tres famílies (veure Figura 41.1). Així, en el nostre disseny podrem posar un Cortex més potent o menys segons l'aplicació o les necessitats sense haver de canviar el dibuix de la PCB. El mateix passa amb altres encapsulats i models tant en aquesta fabricant com a d'altes.

Com que actualment les biblioteques que ofereixen són compatibles entre diferents famílies (o si més no, molt similars) la transició entre diferents famílies acostuma a ser força senzill. La biblioteca **emib** no presenta canvis entre diferents famílies de Cortex, per tant el nostre codi no haurà de recollir cap canvi en aquest sentit.

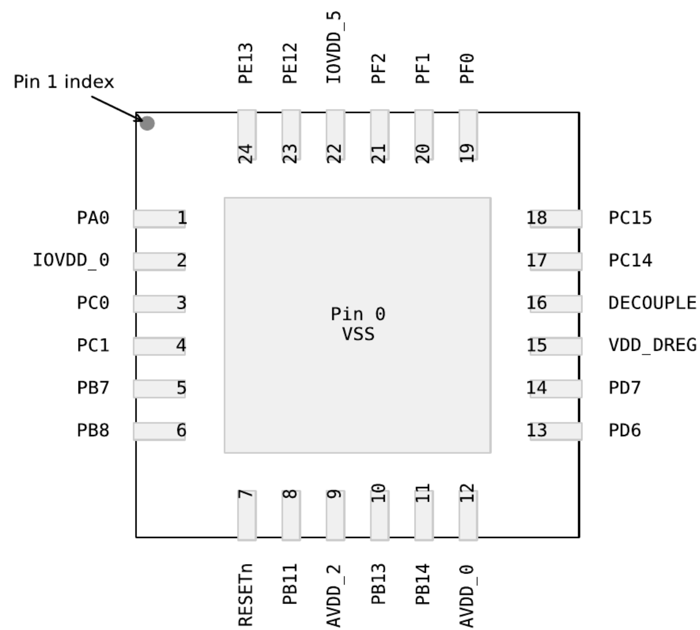


Figura 41.1: Pinout pels microcontroladors
Pinout pels microcontroladors EFM32ZG, EFM32HG i EFM32TG amb encapsulat QFN24 (extret de [94, pàgina 72]).

42. Inicialització del sistema i del llenguatge C

És la funció main() realment la primera funció que s'executa quan comença l'execució? Qui implementa les funcions malloc()/free()? Ara toca aprendre sobre els interiors del runtime de C i com s'inicialitza tot el sistema

Abans no comenci l'execució del nostre programa s'executen tot de funcions per preparar tant el microcontrolador com l'entorn d'execució de C.

Comencem per l'inici: quan el microcontrolador surt de l'estat de reset, el que fa és anar a executar el ResetHandler que està a l'adreça per defecte del Program Counter (registre pc).

Aquesta funció la trobem definida al fitxer startup_gcc_efm32tg.s al directori CMSIS del projecte i aquest handler tant sols crida la funció SystemInit(), tal com es veu a la Figura 42.1.

Podem dir-li al debugger que s'aturi en aquesta funció canviant-li la configuració i dient-li que s'aturi a la funció que vulguem, en aquest cas hi podem escriure Reset_Handler. Per defecte veurem que està configurat per que s'aturi a la funció main() (Figura 42.2).

Aquesta funció està definida pel fabricant i la trobem al fitxer system_efm32tg.c al mateix directori. En el cas dels Cortex-M el que fa és modificar el registre VTOR de la CPU per a que apunti a la taula de vectors d'interrupció definits al fitxer startup_gcc_efm32tg.s.

```
    .globl      Reset_Handler
    .type       Reset_Handler, %function
Reset_Handler:
#ifndef __NO_SYSTEM_INIT
    ldr     r0, =SystemInit
    blx     r0
#endif
```

Figura 42.1: Reset Handler per Cortex-M

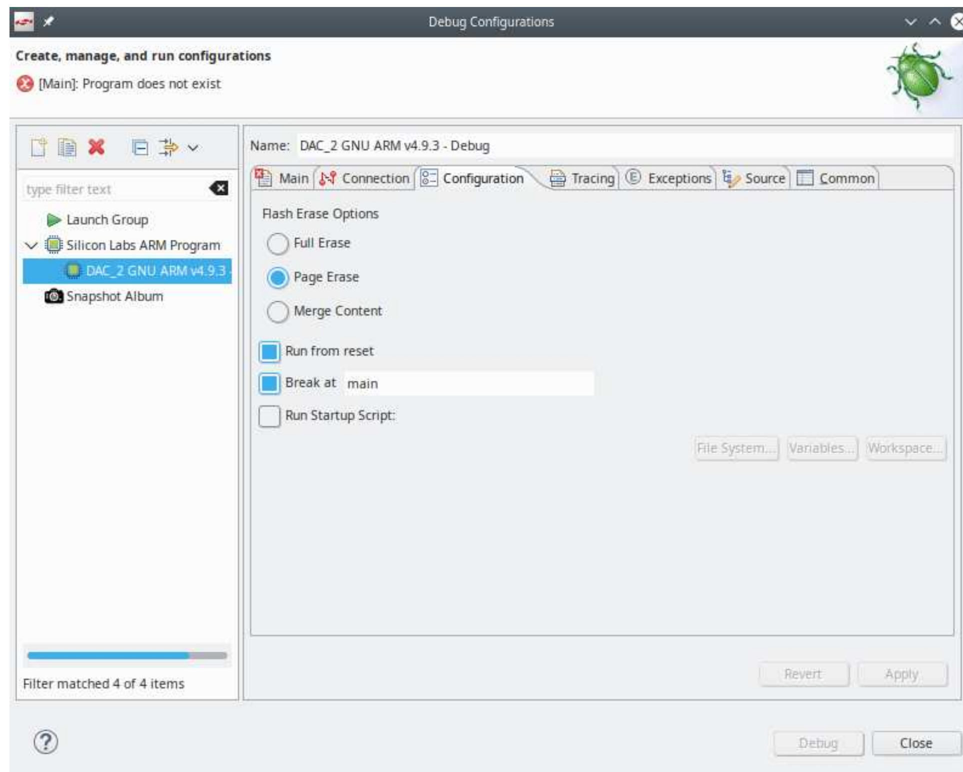


Figura 42.2: Configuració del *Debugger*

A continuació segueix executant-se el Reset Handler, i el primer que fa és copiar la secció **.data** a la RAM (Figura 42.3). Això què vol dir? Doncs que les variables que s'han inicialitzat amb algun valor inicial al nostre codi s'han emmagatzemat al fitxer binari a continuació del codi (secció **.text**). Abans de començar a funcionar el codi, cal copiar aquestes variables a la memòria RAM, que és la secció **.data**. Així, quan aquesta part de la inicialització acaba, tenim les variables a la memòria RAM amb els seus valors inicials.

Un acabada aquesta còpia, es crida a la funció _start() de la biblioteca que estiguem fent servir. En el cas de EFM32 la biblioteca és la Nano C library. Aquesta biblioteca implementa les llibreries estàndard de C (stdlib, string, memory, etc.) i li cal una inicialització que es troba al fitxer crt0.S (situada a newlib/libc/sys/arm/crt0.S).

El que es fa aquí és inicialitzar el punter de l'stack a l'adreça que s'indiqui al fitxer del linker i posar a zero tota la memòria de la secció **.bbs** (Figures 42.4 i 42.5).

A continuació es crida la funció __libc_init_array() (situada a newlib/libc/misc/init.c) que va cridant funcions d'inicialització de la pròpia biblioteca (i constructors estàtics si treballem en C++).

Finalment, la funció _start() crida a la funció main() del nostre programa i ja comença a executar-se el nostre codi (Figura 42.6).

Com es pot veure, no és trivial engegar un microcontrolador i tenir l'entorn del llenguatge C preparat, però per sort tenim aquestes biblioteques i els fitxers que ens donen els fabricants per fer-ho sense que ens n'haguem de preocupar.

```
/*  Single section scheme.
 *
 *  The ranges of copy from/to are specified by following symbols
 *    __etext: LMA of start of the section to copy from. Usually end of text
 *    __data_start__: VMA of start of the section to copy to
 *    __data_end__: VMA of end of the section to copy to
 *
 *  All addresses must be aligned to 4 bytes boundary.
 */
    ldr     r1, =__etext
    ldr     r2, =__data_start__
    ldr     r3, =__data_end__

.L_loop1:
    cmp     r2, r3
    ittt    lt
    ldrlt   r0, [r1], #4
    strlt   r0, [r2], #4
    blt     .L_loop1
#endif /* __STARTUP_COPY_MULTIPLE */
```

Figura 42.3: Còpia de la secció .bss a la memòria RAM

```
    /*  Set up the stack pointer to a fixed value. */
    /*  Changes by toralf:
        - Allow linker script to provide stack via __stack symbol - see
          defintion of .Lstack
        - Provide "hooks" that may be used by the application to add
          custom init code - see .Lhwinit and .Lswinit.  */

    ldr r3, .Lstack
    cmp r3, #0
    /* Note: This 'mov' is essential when starting in User, and ensures we
         always get *some* SP value for the initial mode, even if we
         have somehow missed it below (in which case it gets the same
         value as FIQ - not ideal, but better than nothing).  */
    mov sp, r3

    /* We don't care of r2 value in standalone.  */
    bl FUNCTION (_stack_init)
```

Figura 42.4: Inicialització del registre d'stack (Stack Pointer)

```
/* Zero the memory in the .bss section.  */
movs    a2, #0          /* Second arg: fill value.  */
mov fp, a2          /* Null frame pointer.  */
mov r7, a2          /* Null frame pointer for Thumb.  */

ldr a1, .LC1        /* First arg: start of memory block.  */
ldr a3, .LC2
subs    a3, a3, a1      /* Third arg: length of block.  */
```

Figura 42.5: Inicialització de la secció .bss

```
	ldr	r0, .Latexit
	cmp	r0, #0
	beq	.Lweak_atexit
#endif
	ldr	r0, .Lfini
	bl	FUNCTION (atexit)
.Lweak_atexit:
	bl	FUNCTION (_init)
	movs	r0, r4
	movs	r1, r5
#endif
	bl	FUNCTION (main)

	bl	FUNCTION (exit)		/* Should not return.  */
```

Figura 42.6: Crida a la funció _init() i funció main()

43. Treballant amb punt flotant

Sempre s'ha dit que cal evitar l'ús de variables en punt flotant (*float* o *double*) en sistemes encastats. Això ve dels temps en que els microcontroladors disponibles no tenien cap unitat hardware de punt flotant i aquestes operacions s'havien de fer per software i això penalitzava moltíssim el rendiment.

Això encara és aplicable per la majoria de casos, tot i que els nous microcontroladors basats en Cortex-M4 o Cortex-M7 poden portar unitats de punt flotant. Anem a veure amb detall una mica com treballar amb les eines i el codi perquè tot funcioni correctament.

Treballarem amb dos exemples molt senzills, que son les funcions **mulf()** i **muld()** (Llistats 43.1 i 43.2), que multipliquen dos valors de tipus *float* i *double* respectivament. Cal recordar que el tipus *float* correspon a un tipus de punt flotant de 32 bits conegut com *single precision* seguint l'standard IEEE 754. *Double* correspon a un tipus de 64 bits conegut com *double precision* del mateix standard [97].

Les biblioteques estàndard de C incorporen funcions per operar amb aquests tipus, i son les que es fan servir per defecte pel compilador si no li donem ordres especials.

Així doncs, si compilem el fitxer mulf.c amb la següent comanda

```
> arm-none-eabi-gcc mulf.c -o- -S -mthumb -mcpu=cortex-m4
```

ens mostrarà per pantalla el codi en assemblador que genera el compilador. Cal fixar-se que els

Llistat 43.1: funció mulf()

```c
/* mulf.c */
float mulf(float a, float b) {
  return a*b;
}
```

Llistat 43.2: funció muld()

```
/* muld.c */
double mulf(double a, double b) {
  return a*b;
}
```

Llistat 43.3: codi assemblador de la funció mulf.c

```
        push    {r7, lr}
        sub     sp, sp, #8
        add     r7, sp, #0
        str     r0, [r7, #4]    @ float
        str     r1, [r7]        @ float
        ldr     r1, [r7]        @ float
        ldr     r0, [r7, #4]    @ float
        bl      __aeabi_fmul
        mov     r3, r0
        mov     r0, r3
        adds    r7, r7, #8
        mov     sp, r7
        @ sp needed
        pop     {r7, pc}
```

flags que li passem al compilador son només que el microcontrolador és un Cortex-M4.

Aquí el que es veu és que es preparen uns registres i es crida una funció anomenada __eabi_fmul que és la funció de la biblioteca encarregada de fer les multiplicacions per software.

Si ara especifiquem que el cortex-M4 te el mòdul d'operacions en punt flotant amb la següent comanda:

```
> arm-none-eabi-gcc muld.c -o- -S -mthumb -mcpu=cortex-m4
   -mfloat-abi=hard -mfpu=fpv4-sp-d16
```

El resultat serà el següent, on ja es veu que es fan servir instruccions de punt flotant (vstr, vldr, vmul, vmov, etc.)

En aquest cas els *flags* del compilador indiquen quin mòdul FPU te el nostre microcontrolador (fpv4-sp-d16): fp versió 4, single precision i 16 registres).

Això seria pel cas de la funció que treballa amb precisió simple, si ara fem el mateix per la funció que treballa amb dobles, fent servir els mateixos *flags*

```
> arm-none-eabi-gcc muld.c -o- -S -mthumb -mcpu=cortex-m4
   -mfloat-abi=hard -mfpu=fpv4-sp-d16
```

Veiem que altre cop es fa l'operació per software enlloc de fer-la via les instruccions de punt flotant. Per què passa això? Doncs perquè l'arquitectura Cortex-M4 només permet FPUs de precisió simple i no pot treballar amb precisió doble i així li hem especificat al compilador. Per tant el compilador fa les crides a la biblioteca software pertinent (__eabi_dmul).

Per tant, compte amb treballar amb doubles i arquitectures Cortex-M4 o inferiors! Cal tenir en

Llistat 43.4: codi assemblador de la funció mulf.c usant FPU

```
        push     {r7}
        sub      sp, sp, #12
        add      r7, sp, #0
        vstr.32 s0, [r7, #4]
        vstr.32 s1, [r7]
        vldr.32 s14, [r7, #4]
        vldr.32 s15, [r7]
        vmul.f32         s15, s14, s15
        vmov.f32         s0, s15
        adds     r7, r7, #12
        mov      sp, r7
        @ sp needed
        ldr      r7, [sp], #4
        bx       lr
```

Llistat 43.5: codi assemblador de la funció muld.c

```
        push     {r7, lr}
        sub      sp, sp, #16
        add      r7, sp, #0
        vstr.64 d0, [r7, #8]
        vstr.64 d1, [r7]
        ldrd     r2, [r7]
        ldrd     r0, [r7, #8]
        bl       __aeabi_dmul
        mov      r2, r0
        mov      r3, r1
        vmov     d7, r2, r3
        vmov.f32         s0, s14
        vmov.f32         s1, s15
        adds     r7, r7, #16
        mov      sp, r7
        @ sp needed
        pop      {r7, pc}
```

Llistat 43.6: codi assemblador de la funció muld.c usant FPU de Cortex-M7

```
push     {r7}
sub      sp, sp, #20
add      r7, sp, #0
vstr.64 d0, [r7, #8]
vstr.64 d1, [r7]
vldr.64 d6, [r7, #8]
vldr.64 d7, [r7]
vmul.f64         d7, d6, d7
vmov.f64         d0, d7
adds     r7, r7, #20
mov      sp, r7
@ sp needed
ldr      r7, [sp], #4
bx       lr
```

compte que algunes operacions en C que fan servir constants poden acabar en un tipus double si no vigilem.

Si ara fem la prova amb la mateixa funció però usant els *flags* per un Cortex-M7

```
> arm-none-eabi-gcc muld.c -o- -S -mthumb -mcpu=cortex-m7
    -mfloat-abi=hard -mfpu=fpv5-d16
```

En aquest cas, els *flags* indiquen que el processador és un Cortex-M7 i la unitat de punt flotant és la versió 5 (Tal com indica el ARM Cortex-M7 Processor Technical Reference Manual [98][8-2)].

Amb aquests *flags*, el codi assemblador que es genera és el que es veu al llistat següent

Aquí es veu que es torna a fer servir registres i instruccions pròpies del punt flotant (vstr, vldr, vmul, vmov) amb el suffix .f64 que es correspon a la mida del tipus double (64 bits) i que, per tant, les operacions no es fan per una rutina software si no que les executa el mòdul de punt flotant del microcontrolador.

Per tant, podem veure clar que l'ús del tipus double només és recomanat per arquitectures Cortex-M7 i posteriors si no volem tenir una pèrdua de rendiment considerable.

Per últim, no cal espantar ningú, ja que aquest *flags*> els maneguen les eines de cada fabricant segons les característiques dels seus microcontroladors i normalment no ens n'hem de preocupar.

Bones pràctiques

44 **Ús de memòria dinàmica** **247**

45 **Ús de *volatile*** **249**

46 **Funcions re-entrants** **251**

47 ***Deadlock*** **253**

48 **Inversió de prioritats** **255**

49 **Assignació de prioritats** **257**

50 **Mida de les cues** **259**

50.1 Model M/M/1

51 ***Debounce*** **263**

51.1 Un exemple de *debouce*

52 **Ús eficient de printf** **267**

53 **Empaquetant estructures** **269**

53.1 Un exemple senzill

Aquesta pàgina està en blanc expressament, tot va bé.

En aquest capítol veurem una sèrie de bones pràctiques habituals en la programació de sistemes encastats. Aquestes bones pràctiques donen consells i guia sobre com dissenyar o programar parts de codi per evitar problemes que, habitualment, són molt complicats de detectar.

Aquesta pàgina està en blanc expressament, tot va bé.

44. Ús de memòria dinàmica

Una de les diferències més notables a l'hora d'escriure codi per un sistema encastat és l'ús de memòria dinàmica. Bàsicament se'n desaconsella totalment el seu ús en sistemes encastats. Això es deu al fet que tenim molt poca memòria RAM disponible (pocs KB) i que la possible fragmentació que s'origina en fer-ne un ús dinàmic poc exhaurir-la molt més fàcilment. A més, el fet d'usar memòria dinàmica fa que el sistema sigui menys predictible, ja que en certs casos, l'ordre en que s'executen diferents *malloc()* pot ser diferent a cada execució.

És per això que no s'acostuma a usar memòria dinàmica en sistemes encastats. Si, tot i la recomanació de no fer-ho, és necessari alguna mena de gestió dinàmica de la memòria, la millor opció és proveir-se d'una estructura pròpia anomenada *pool* de blocs d'una mida predeterminada que proporcionin aquesta funcionalitat. D'aquesta manera s'evita la fragmentació ja que tots els blocs tenen la mateixa mida.

En alguns casos, pot ser inevitable l'us de memòria dinàmica (inicialització d'estructures que no se sap a priori si caldran o no) i és acceptable fer aquesta reserva de memòria en el moment d'inicialització del sistema.

Així, podriem resumir que el que està prohibit és l'ús de la comanda **free()** més que no pas la comanda **malloc()**.

Aquesta pàgina està en blanc expressament, tot va bé.

45. Ús de *volatile*

Com ja s'ha comentat a **Subsecció 7.2.1 - Ús de variables globals**, errors en l'ús de la paraula reservada *volatile* poden ocasionar *bugs* difícils de trobar al nostre codi. Per tant, i com a recordatori, cal definir una variable com a *volatile* en el següents casos:

- Variable global que comunica una ISR amb una funció.
- Variable comptador d'un bucle per implementar un *delay*.
- Punter a una adreça de memòria corresponent a un perifèric mapat a memòria.
- Variable global que hi accedeixen dues o més tasques d'un RTOS.

Cal recordar que l'ús de *volatile* farà que les optimitzacions del compilador no s'apliquin a la variable definida com a tal.

Aquesta pàgina està en blanc expressament, tot va bé.

46. Funcions re-entrants

Com ja es va comentar breument a **Secció 27.2 - Modificant el *wrapper* d'I2C**, quan es treballa en un entorn multitasca (com quan es té un RTOS) cal tenir en compte que funcions que puguin ser utilitzades alhora per més d'una tasca cal que siguin re-entrants. També cal adonar-se que una biblioteca per un perifèric HW qualsevol segurament haurà de ser re-entrant, ja que diverses crides simultànies sobre el mateix HW pot ocasionar errors de funcionament.

La norma general és de protegir cada funció que hagi de ser re-entrant amb un *Mutex*. La funció en qüestió intentarà agafar el *Mutex* a l'inici de la seva execució i el retornarà en quan acabi. En el cas de biblioteques per accedir a HW, és habitual tenir un sol *Mutex* compartit per tota la biblioteca i que es crea quan es crida a la funció d'inicialització de la biblioteca (veure **Secció 27.2 - Modificant el *wrapper* d'I2C**).

Aquesta pàgina està en blanc expressament, tot va bé.

47. Deadlock

Un *Deadlock* és una situació on diverses tasques tenen una dependència circular entre elles i queden totes elles bloquejades esperant-se unes a les altres.

Per evitar aquestes situacions, sovint complexes de detectar, hi ha dues recomanacions:

- Evitar adquirir dos o més *Mutex*. Provar d'agafar dos o més *Mutex* pot provocar que s'agafi un però fallin els demès, fent que la tasca hagi d'esperar a d'altres tasques els alliberin, que potser necessiten del primer *Mutex*.
- Ordenar els *Mutex* de manera que, si s'ha d'agafar més d'un, totes les tasques segueixin el mateix ordre.

Amb aquests dues recomanacions es poden evitar la majoria de *deadlocks* generats per l'ús de *mutex* entre tasques.

Aquesta pàgina està en blanc expressament, tot va bé.

48. Inversió de prioritats

Quan tenim un parell de tasques que comparteixen un recurs, una amb poca prioritat (T_l) i la segona amb més prioritat (T_h), si s'afegeix una tercera tasca amb una prioritat intermèdia (T_m) al sistema, podem tenir un problema d'inversió de prioritats. Això passarà quan la tasca de menys prioritat agafa el recurs compartit amb (T_h). En aquest moment, si la tasca de prioritat intermèdia està a l'estat *Ready*, passarà a executar-se, fent que la tasca (T_l) no s'executi i retardant l'execució de la tasca (T_h), fent que, de fet, la prioritat de T_h i T_m s'hagin invertit, ja que la tasca amb prioritat intermèdia es pot executar tot el temps que vulgui i la tasca més prioritària no té la oportunitat [99, pàgina 101].

La manera més senzilla de resoldre aquest problema és usar *Mutex* amb herència de prioritats. Aquest mecanisme fa que, provisionalment, la tasca que agafa el *Mutex* pugi temporalment la seva prioritat a la mateixa de la tasca que l'està esperant [99, pàgina 106]. FreeRTOS suporta aquest mecanisme als seus *Mutex*, i per tant fent un bon ús dels mateixos evitarem aquest fenomen d'inversió [52, pàgina 251].

Aquesta pàgina està en blanc expressament, tot va bé.

49. Assignació de prioritats

Sovint un dels dubtes que sorgeixen en el disseny de sistemes encastats és quines prioritats cal donar a cada una de les tasques del sistema. Existeix un algorisme molt senzill per assignar les prioritats a cada tasca, basant-se en el temps de procés que necessita cada una. Aquest algorisme s'anomena *Rate-Monotonic Algorithm* (RMA) i fa les següents assumpcions [100][101, pàgina 136]:

- Totes les tasques són periòdiques.
- El *deadline* de cada tasca és el seu període.
- Totes les tasques són independents.
- Totes les tasques són pre-emtives i el cost d'aquest és negligible.

Aquest algorisme senzillament assigna la prioritat més alta a les tasques amb un període més curt. Així, s'ordenen les tasques segons el seu període (primer els períodes més curts) i s'assignen les prioritats, de més alta a més baixa.

Per saber si es podran executar totes les tasques dins dels seus límits complint tots els *deadline* es poden fer els següents càlculs:

Sigui c_i el temps d'execució de la tasca T_i. Sigui p_i el període d'execució de la tasca T_i. Sigui n el nombre de tasques totals. Es defineix l'ús acumulat μ a:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i}$$

S'ha de complir la condició 49.1 perquè es compleixin tots els *deadlines* de totes les tasques, sempre amb els supòsits inicials.

$$\mu \leq n(2^{1/n} - 1) \tag{49.1}$$

Així, si tenim 3 tasques amb les dades d'execució de la Taula 49.1 l'algorisme RMA assignarien les prioritats de la següent manera:

1. T3 més prioritària.
2. T2 prioritat intermèdia.
3. T1 baixa prioritat.

També es pot calcular μ

$$\mu = \frac{20}{500} + \frac{30}{250} + \frac{15}{100} = 0.31$$

i segons l'Equació 49.1 tindrem que

$$0.31 \leq n(2^{1/n} - 1) = 3(2^{1/3} - 1) \approx 0.78$$

Per tant es compleixen les condicions perquè les tres tasques es puguin executar sense perdre cap esdeveniment. L'algorisme RMA dona una conjunt de prioritats que és òptima, per tant, si no es compleixen els *deadlines*, cap altre mètode d'assignar prioritats fixes podrà aconseguir-ho. En aquest cas caldrà tenir un *scheduler* amb un algorisme de prioritats dinàmiques.

També val la pena observar que la part dreta de l'Equació 49.1 té un límit:

$$\lim_{n\to\infty} n \cdot (2^{1/n} - 1) = ln(2) \approx 0.7$$

que ens indica que amb les condicions dites abans, un sistema amb moltes tasques hauria de dedicar el 70% d'ocupació total per garantir tots els *deadlines* de les tasques.

Taula 49.1: Dades d'exemple de tasques i prioritats (temps en mil·lisegons)

Tasca	**Període** p	**Temps d'execució** c
T1	500	20
T2	250	30
T3	100	15

50. Mida de les cues

Quan hem parlat de les cues en un RTOS a **Secció 25.2 - Cues**, hem dit que a l'hora de la seva creació cal especificar el tipus de dades que emmagatzemarà cada element de la mateixa i el nombre d'elements d'aquest tipus que la cua manegarà.

Però, com saber quants elements cal atorgar a una cua en la seva creació? Aquest paràmetre serà clau, ja que si creem una cua amb pocs elements disponibles, la tasca productora potser es quedi bloquejada si la tasca consumidora no va prou de pressa. Tot i que es pot triar aquest valor d'una forma empírica, començant per un valor prou baix i fent proves i via successives aproximacions arribar a un valor prou bo.

Aquest mètode, però, no ens assegura que en qualsevol cas el sistema no acabi amb una cua plena. Per això, cal un anàlisi més analític del problema per trobar una solució.

50.1 Model M/M/1

Aquest model de cues és dels models estadístics més senzills però que ens pot donar informació important només amb les dades més bàsiques del nostre sistema. Aquest model fa certes suposicions que podem donar per bones pels nostres sistemes [102][103][104][105]:

- El productor genera noves entrades a la cua seguint una distribució de Poisson.
- El consumidor processa dades a la cua seguint una distribució exponencial.
- Només hi ha un productor.
- La cua és de tipus FIFO.

Amb aquestes suposicions ens cal trobar els paràmetres λ i μ pel productor i el consumidor respectivament, ara es veurà com.

Si la nostra tasca consumidora genera un element nou a la cua de mitjana (seguint una distribució de Poisson) cada cert Pr temps tindrem:

$$Pr = \text{temps mitjà a generar una dada} \tag{50.1}$$

i llavors tindrem que

$$\lambda = \frac{1}{Pr} \tag{50.2}$$

El mateix càlcul el podem fer pel temps de la tasca consumidora (que segueix una distribució exponencial):

$$C = \text{temps mitjà a processar una dada} \tag{50.3}$$

i llavors tindrem que

$$\mu = \frac{1}{C} \tag{50.4}$$

Amb aquestes dades, tenim les següents fórmules:

$$\rho = \frac{C}{Pr} = \frac{\lambda}{\mu} \tag{50.5}$$

Aquesta primer valor ρ ens indica si el sistema és factible o no: si ρ és més petit d'1 ($\rho < 1$), la cua té sentit, en cas contrari, el ritme de inserir elements a la cua és més ràpid que el ritme de treure'ls i, per tant. la cua s'acabarà omplint en algun moment o altre i el productor haurà de llençar dades que no podrà inserir a la cua.

Amb aquest valor ρ (o amb Pr i C) podem obtenir els següents càlculs:

Nombre mitjà d'elements a la cua

$$L_q = \frac{\rho^2}{(1-\rho)} = \left(\frac{C}{Pr}\right)^2 / \left(1 - \frac{C}{Pr}\right) \tag{50.6}$$

Temps mitjà de vida a la cua

$$W_q = \frac{\rho}{\mu - \lambda} = \frac{L_q}{\lambda} = L_q \cdot Pr \tag{50.7}$$

Temps total d'estada en el sistema (procés més espera a la cua)

$$W = W_q + \frac{1}{\mu} = W_q + C = \frac{C}{1-\rho} \tag{50.8}$$

Nombre mitjà d'elements al sistema

$$L = \frac{\rho}{(1-\rho)} = W \cdot \lambda = \frac{W}{Pr} \tag{50.9}$$

Probabilitat que la cua tingui més de K elements

$$P(\geqslant K) = \rho^K = \left(\frac{C}{Pr}\right)^k \tag{50.10}$$

Així si, per exemple, tenim una tasca productora que genera una dada cada 50 ms i una tasca consumidora que processa una dada en uns 30 ms de mitjana, tenim els següents resultats:

$$Pr = 50 \text{ ms}$$

$C = 30$ ms

$\rho = \frac{C}{Pr} = \frac{30}{50} = 0.6$

Nombre mitjà d'elements a la cua $L_q = \left(\frac{30}{50}\right)^2 / \left(1 - \frac{30}{50}\right) = 0.9$

Temps mitjà de vida a la cua $W_q = 0.9 \cdot 50 = 45$ ms

Temps total de vida d'una dada $W = \frac{30 \cdot 50}{50 - 30} = 75$ ms

Nombre mitjà d'elements al sistema $L = \frac{75}{50} = 1.5$

Probabilitat que la cua tingui més de 10 elements $P(\geqslant 10) = \left(\frac{30}{50}\right)^{10} \approx 0,00605 \rightarrow 0.60\%$

Aquestes equacions ens indiquen que durant bona part del temps de funcionament del sistema, la cua entre els dos processos tindrà tant sols 1 element, i que la probabilitat que tingui més de 10 elements en algun moment és de només el 0,60%. Cal fer notar que aquest valor probabilístic té en compte que els processos que generen dades es comporten com una variable aleatòria tipus Poisson i els temps de processat les dades s'ajusta a una variable aleatòria exponencial. Si algun dels dos processos no es comporta com a tal, si no que el seu temps de procés o de generació de dades és fix, els valors L_q, W_q, W i L seran certs en tot moment.

Manipulant una mica les fórmules, també podem esbrinar quin temps màxim de procés podem tenir per una tasca que genera dades cada 25 ms i volem menys d'un 0.1% de probabilitats que la cua arribi a tenir 8 elements.

Tenim, doncs:

$Pr = 25$ ms

$K = 8$

Segons la fórmula 50.10:

Probabilitat que la cua tingui més de K elements $= \rho^K = (\frac{C}{Pr})^K < 0.001(0.1\%)$

per tant tenim que

$C^8 < 25^8 * 0,001 \rightarrow C < \sqrt[8]{25^8 * 0,001} \approx 10.54$ ms

Això ens indica que el temps de processar una dada per part el consumidor (C) ha de ser menor de 10.54 mil·lisegons de mitjana per assegurar els requeriments donats.

Aquesta pàgina està en blanc expressament, tot va bé.

51. *Debounce*

Un problema que ens podem trobar quan volem llegir una entrada digital, és el fenomen dels rebots: si el pin està connectat a un botó a algun altre accionador mecànic aquest pot generar rebots al senyal, que vol dir que no es genera un pols quadrat i perfecte si que no quan es genera un pols aquest vagi acompanyat per d'altres polsos més petits i espuris. Habitualment les sortides d'altres components digitals no presenta aquest fenomen i no cal fer servir aquestes tècniques.

Aquest efecte pot provocar que el nostre codi compti més polsos dels que realment s'haurien de comptar i tenir un sistema erroni.

Per solucionar-ho, a part d'afegir certa circuiteria addicional al voltant del pin d'entrada, es pot desenvolupar codi que tingui en compte aquesta situació. Aquesta mena de codis es coneixen com *debouncing* i normalment es basen en llegir vàries vegades el pin implicat i veure quan deixa de canviar i amb això decidir si hi hagut canvi en el valor del pin o no.

Aquests algorismes han de decidir el més ràpid possible si l'entrada ha canviat o no i per contra quan més temps estiguin avaluant l'entrada millor funcionaran i detectaran espuris (*glitches*). A més, quan més cops per segons s'avalua el valor d'un pin més ocupació e la CPU es tindrà per aquesta tasca.

Les tècniques més habituals es basen en programar un *timer* o una tasca programada per que cridi una funció d'avaluació de forma periòdica (cada X mil·lisegons) i la dita funció llegeixi el valor de l'entrada i decideixi el valor real de l'entrada [106][107].

Un altre forma de fer-ho, potser més senzilla és la de un cop detectat un primer flanc, deixar de llegir l'entrada fins passat un temps i un cop transcorregut el temps, es llegeix el valor de l'entrada altre cop. Això es pot fer fàcilment controlant un Timer des de la ISR d'entrada del pin, tal com es veurà a continuació.

Llistat 51.1: ISR del timer per fer debouncing

```
void TIMER1_IRQHandler(void) {
  uint32_t flags;

  /* Clear flag for TIMER1 */
  flags = TIMER_IntGet(TIMER1);
  TIMER_IntClear(TIMER1, flags);

  timer_running = false;

  if (GPIO_PinInGet(gpioPortD, 8) == 1) {
    button_counter++;
  }
}
```

Llistat 51.2: Codi per engegar el timer a la ISR del GPIO

```
void GPIO_EVEN_IRQHandler(void) {
  ...
  if (!timer_running) {
    timer_running = true;
    TIMER_TopSet(TIMER1, DEBOUNCE_VALUE);
    TIMER_Enable(TIMER1, true);
  }
...
```

51.1 Un exemple de *debouce*

El codi d'aquest examples està, com sempre, al repositori. Primer cal configurar el Timer per què compti un cert temps i generi una IRQ un cop transcorregut aquest temps. Per això configurem el valor Top tal com ja vàrem fer a **Capítol 8 - Timers**.

En aquest exemple es configura el valor top per que estigui comptant 100 mil·lisegons fent un càlcul molt similar al de l'exemple amb Timers anterior. També es prepara la ISR pel *Timer1* tal com es veu al Llistat 51.1 (veure Figura 51.1).

La variable *timer_running* es defineix com una variable booleana (i volàtil) amb valor per defecte a false. A aquesta ISR es comprova el valor desitjat de l'entrada i si és el cas, s'actualitza el comptador.

Per últim a la ISR del GPIO corresponent inserim el codi següent per engegar el *Timer* quan es detecti un flanc al senyal (un canvi al seu valor), tal com es veu al Llistat 51.2:

D'aquesta manera tant senzilla evitarem els molests rebots i, de fet, tindrem filtrats tots els polsos que considerem massa ràpids pel nostre sistema.

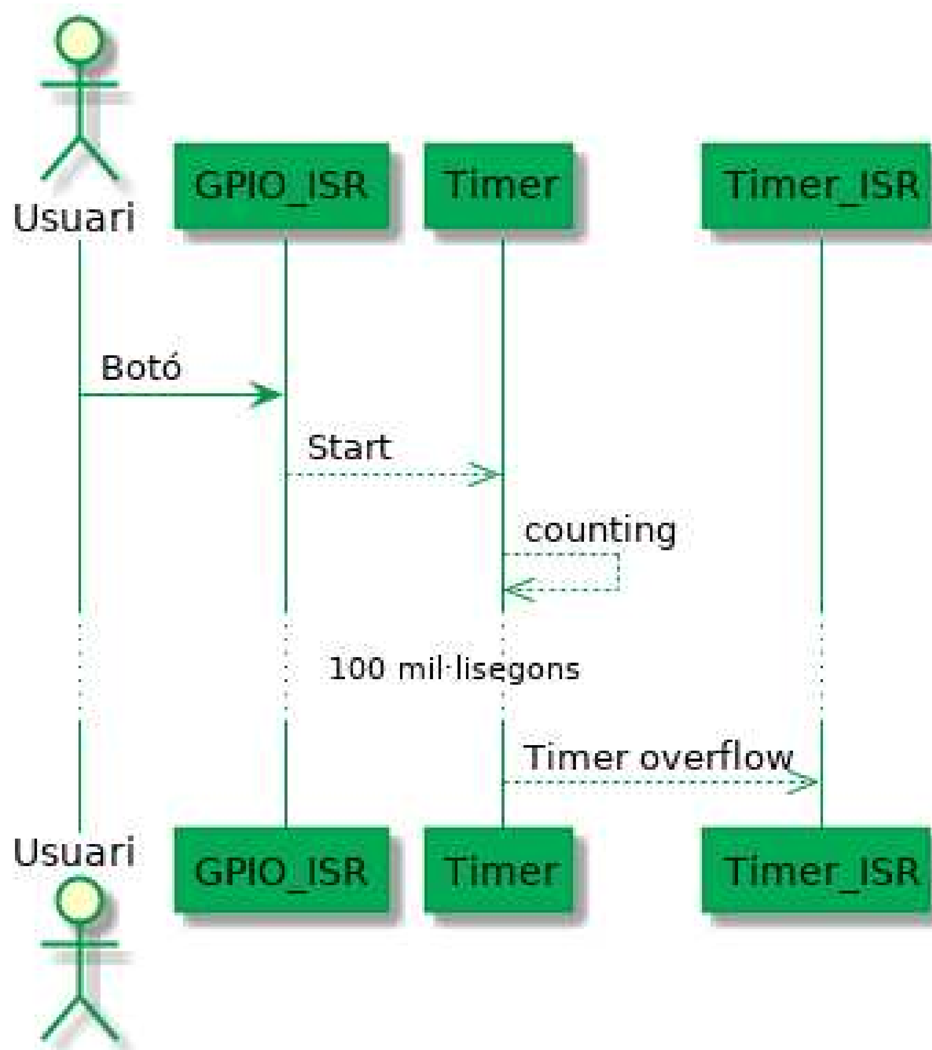


Figura 51.1: Diagrama de seqüència de l'exemple de *debounce*

Aquesta pàgina està en blanc expressament, tot va bé.

52. Ús eficient de printf

Com ja es va veure a **Capítol 4 - Fent servir printf** és possible tenir la funció **printf**() en els nostres sistemes encastats, pagant el preu de gastar força memòria FLASH per la seva implementació.

Una opció recomanable en cas que l'ocupació de la memòria FLASH pugui ser un problema, és el de tenir diferents versions de **printf**() segons els paràmetres que pot rebre. Així, enlloc de tenir el **printf**() genèric de la biblioteca que accepta tot de tipus de tipus de dades segons el format tindrem una funció per imprimir un enter en decimal, una altra per imprimir un enter en hexadecimal, una funció per imprimir una cadena, etc. com es pot veure al Llistat 52.1

A més, totes aquestes noves funcions les usarem a través d'una *macro* de C, de forma que quan passem a una compilació de *release* del projecte aquests **printf**() desapareguin del nostre codi.

Llistat 52.1: Diferents implementacions de **printf()**

```
void printf_char(char ch) {
  ITM_SendChar(ch);
}

void printf_string(char* str) {
  int i = 0;
  while(str[i]) {
    printf_char(str[i]);
    i++;
  }
}

void printf_hex8(uint8_t val) {
  if ((val >>4) > 9) {
    printf_char((val>>4) + '0' + 7);
  } else {
    printf_char((val>>4) + '0');
  }
  if ((val&0x0F) > 9) {
    printf_char((val&0x0F) +'0' + 7);
  } else {
    printf_char((val&0x0F) +'0');
  }
}

...

void printf_int(int val) {
  int rem_dec;
  int dec;
  int i;
  char buffer[10];

  i = 0;

  if (val < 0) {
    printf_char('-');
    val = -1 * val;
  }

  dec = val;
  rem_dec = val;

  do {
    rem_dec = dec%10;
    dec /= 10;
    buffer[i] = '0'+rem_dec;
    i++;
  } while(dec > 10);
  buffer[i] = '0' + dec;

  /* print reverse buffer */
  for(; i >= 0; i--) {
    printf_char(buffer[i]);
  }
}
```

53. Empaquetant estructures

L'ús d'estructures (*struct* en C) per emmagatzemar dades que estan relacionades és força habitual. Per fer-ho, només cal definir una estructura i cada camp es defineix amb el tipus desitjat. Tota l'estructura funciona com un paquet de dades, que es pot moure, copiar i accedir com un tot.

Però si volem accedir a baix nivell a aquestes estructures per, per exemple, enviar les dades que conté per un port sèrie, inserir-la a un paquet de xarxa o enviar-ho a un altre dispositiu via SPI o I2C, cal que tinguem compte el problema de l'empaquetament.

Quan definim una estructura en C, el compilador ha de decidir com l'emmagatzema a la memòria. Segons les característiques dels busos i l'arquitectura del microcontrolador, pot ser que els accessos a memòria només es puguin fer a nivell de paraula (en el cas d'ARM una paraula és de 32 bits) i que no es pugui accedir a un byte individual de la memòria.

I com afecta això a les estructures? Doncs que el compilador pot optar a col·locar els diferents camps de l'estructura ocupant cada un una posició de memòria enlloc d'empaquetar-los tant com pugui.

Així, si tenim una estructura definida com es veu al Llistat 53.1 el compilador guardarà l'estructura a la memòria tal com es veu a la Figura 53.3.

Llistat 53.1: Estructura d'exemple

```
 struct {
        uint8_t fieldS1;
        uint16_t fieldS1b;
        uint32_t fieldL1;
        uint32_t fieldL2;
        uint8_t fieldS2;
} unpacket_struct;
```

Byte 3	Byte 2	Byte 1	Byte 0	Adreça
fieldS1b[1]	fieldS1b[0]	0	fieldS1	**0x00020000**
fieldL1[3]	fieldL1[2]	fieldL1[1]	fieldL1[0]	**0x00020004**
fieldL2[3]	fieldL2[2]	fieldL2[1]	fieldL2[0]	**0x00020008**
0	0	0	fieldS2	**0x0002000A**
				0x0002000C

Figura 53.1: Disposició de l'estructura a la memòria

Llistat 53.2: Estructura d'exemple empaquetada

```
struct __attribute__ ((__packed__)) {
        uint8_t fieldS1;
        uint16_t fieldS1b;
        uint32_t fieldL1;
        uint32_t fieldL2;
        uint8_t fieldS2;
} packet_struct;
```

Que com es pot veure aquesta organització no és la que ens podríem esperar, ja que el camp **fieldS1b** no està enganxat al camp **fieldS1** i es per una posició de memòria per allà enmig. Aquesta operació s'anomena *padding* i és força habitual en totes les arquitectures. En aquest cas fa que aquesta estructura ocupi 16 bytes a la memòria enlloc dels 12 que podria ocupar si estigues tot ben empaquetat.

Això no s'acostuma a tenir gaire en compte alhora de programar sistemes encastats, però pot ser força important si en algun moment una estructura d'aquest estil cal enviar-la byte a byte a algun mòdul o perifèric. Anem a suposar que enviarem aquesta estructura d'exemple pel port sèrie. Si fem una funció que vagi llegint byte a byte l'estructura, tindrem que llegirà uns buits a 0 enmig que ens esgarraran el resultat.

En aquests casos, cal dir-li al compilador que volem que empaqueti tant com pugui l'estructura. Això és fa amb una comanda pròpia de cada compilador, en el cas de GCC és la comanda **__attribute__** que es fa servir tal com es veu al Llistat 53.2. Amb aquesta comanda l'estructura a memòria queda com es veu a la Figura 53.3.

Fent servir aquest atribut es veu que està tot ben empaquetat i ens estalvia uns quants bytes. A més, s'han omplert tots els forats de manera que ara si que podrem accedir byte a byte l'estructura sense problemes.

Cal dir que en força casos aquestes estructures empaquetades poden ser més lentes d'accedir-hi, ja que la CPU haurà d'accedir a diferents posicions de memòria i reconstruir el valor original movent

Byte 3	Byte 2	Byte 1	Byte 0	Adreça
fieldL1[0]	fieldS1b[1]	fieldS1b[0]	fieldS1	**0x00020000**
fieldL2[0]	fieldL1[3]	fieldL1[2]	fieldL1[1]	**0x00020004**
fieldS2	fieldL2[3]	fieldL2[2]	fieldL2[1]	**0x00020008**
				0x0002000A
				0x0002000C

Figura 53.2: Disposició de l'estructura empaquetada a la memòria

bits amunt i avall (veure per exemple, com es reconstruiran els camps fieldL1 o fieldL2)

53.1 Un exemple senzill

A l'exemple del repositori es defineixen dos estructures iguals, una amb l'atribut per empaquetar-la i l'altra amb les opcions per defecte.

Primer es treuen per la consola les mides de totes dues estructures, que encaixen amb el que hem dit aquí i tot seguit es pinten byte a byte per observar els zeros enmig i com està emmagatzemada cada estructura.

Cal destacar com s'accedeix byte a byte a l'estructura. Es defineix un apuntador a byte (*uint8_t **) i es fa apuntar a l'adreça d'inici de l'estructura que es vol analitzar. Tot seguit es va imprimint byte a byte el contingut de la memòria on està emmagatzemada l'estructura.

També es pot analitzar directament el contingut de la memòria usant l'IDE Simplicity Studio fent servir l'eina de *dump* de la memòria tal com es veu a la Figura 53.3.

Llistat 53.3: Mostrant una estructura *byte* a *byte*

```
...
  uint8_t *buffer;

  buffer = (uint8_t*) &unpacket_struct;
  printf("Unpacket structure: \t");
  for(i = 0; i < sizeof(unpacket_struct); i++) {
          printf("0x%02X, ", buffer[i]);
  }
  printf("\n");
...
```

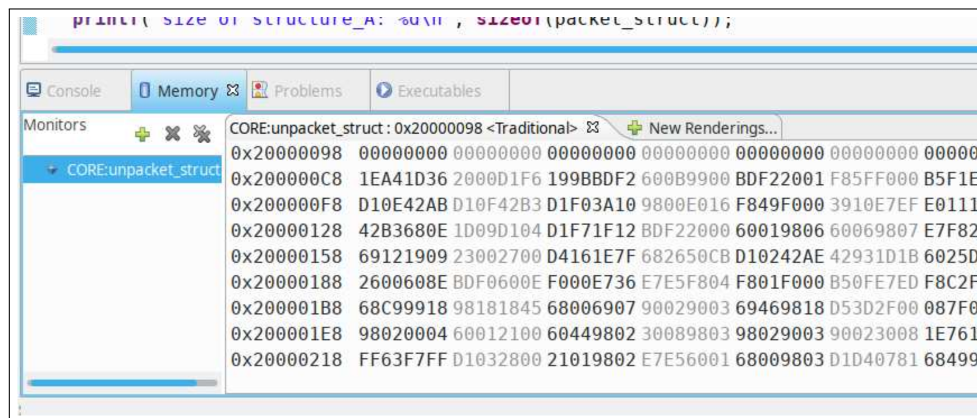


Figura 53.3: Detall de la finestra de *memory dump* a Simplicity Studio

Aquesta pàgina està en blanc expressament, tot va bé.

VIII

Índex, Bibliografia, Glossari

Enllaços dels exemples 277

Bibliografia 279
Llibres

Glossari 287

Acrònims 291

Índex de figures 297

Índex de llistats 299

Índex de funcions 303

Aquesta pàgina està en blanc expressament, tot va bé.

Enllaços dels exemples

- Perifèrics mapats a memòria GitHub
- Prova de velocitat GitHub
- Prova de velocitat 2 GitHub
- Consola Debug GitHub
- GPIO GitHub
- Controlador d'interrupcions GitHub
- Timers GitHub
- Timers 2 GitHub
- Watchdog GitHub
- ADC GitHub
- DMA GitHub
- UART GitHub
- I2C GitHub
- PWM GitHub
- Primer exemple en FreeRTOS GitHub
- Semàfors en FreeRTOS GitHub
- Cues en FreeRTOS GitHub
- Mutex en FreeRTOS GitHub
- Primera aplicació en FreeRTOS GitHub
- Gestió d'execepcions GitHub

Aquesta pàgina està en blanc expressament, tot va bé.

Bibliografia

[1] Free Software Foundation. *Llicència Pública General de GNU*. https://www.gnu.org/licenses/gpl.html (vegeu la pàgina 13).

[2] Arduino. *What is Arduino?* https://www.arduino.cc/en/Guide/Introduction (vegeu la pàgina 14).

[4] Silicon Labs. *EFM32TG Reference Manual*. https://www.silabs.com/documents/public/reference-manuals/EFM32TG-RM.pdf. Silicon Labs (vegeu les pàgines 14, 105, 109, 110, 207, 231 - 233).

[5] Silicon Labs. *SImplicity Studio v4*. http://www.silabs.com/products/development-tools/software/simplicity-studio (vegeu la pàgina 14).

[6] Broadcom. *APDS-9960 Datasheet*. https://www.broadcom.com/products/optical-sensors/integrated-ambient-light-and-proximity-sensors/apds-9960 (vegeu les pàgines 14, 90, 92, 118, 121).

[7] ARM. *GNU Arm Embedded Toolchain*. https://developer.arm.com/open-source/gnu-toolchain/gnu-rm (vegeu la pàgina 16).

[8] ARM. *ARM Processors*. https://www.arm.com/products/processors. ARM (vegeu la pàgina 19).

[9] Silicon Labs. *Silicon Labs Hompage*. https://www.silabs.com/ (vegeu la pàgina 20).

[10] Texas Instruments. *Texas Instruments Hompage*. https://www.ti.com/ (vegeu la pàgina 20).

[11] NXP. *NXP Hompage*. https://www.nxp.com/ (vegeu la pàgina 20).

[12] ST. *ST Homepage*. https://www.st.com/ (vegeu la pàgina 20).

[13] Microchip. *Microchip Hompage*. https://www.Microchip.com/ (vegeu la pàgina 20).

[14] ARM. *Cortex-A Series*. https://www.arm.com/products/processors/cortex-a. ARM (vegeu la pàgina 20).

[15] ARM. *Cortex-R Series*. https://www.arm.com/products/processors/cortex-r. ARM (vegeu la pàgina 20).

[16] ARM. *Cortex-M Series*. https://www.arm.com/products/processors/cortex-m. ARM (vegeu la pàgina 20).

[18] ARM. *Cortex-M0+*. https://developer.arm.com/products/processors/cortex-m/cortex-m0-plus. ARM (vegeu la pàgina 20).

[19] ARM. *Cortex-M3*. https://developer.arm.com/products/processors/cortex-m/cortex-m3. ARM (vegeu la pàgina 20).

[20] ARM. *Cortex-M4*. https://developer.arm.com/products/processors/cortex-m/cortex-m4. ARM (vegeu la pàgina 20).

[21] ARM. *Cortex-M7*. https://developer.arm.com/products/processors/cortex-m/cortex-m7. ARM (vegeu la pàgina 20).

[24] Silicon Labs. *EFM32G Reference Manual*. http://www.silabs.com/documents/public/reference-manuals/EFM32G-RM.pdf. Silicon Labs (vegeu les pàgines 22, 23, 37, 48, 51, 53, 58, 62, 69, 76, 79, 94, 206).

[25] Silicon Labs. *EMLIB library*. https://siliconlabs.github.io/Gecko_SDK_Doc/efm32tg/html/group__emlib.html (vegeu les pàgines 24, 51).

[26] ST. *STM32 Standard Peripheral Libraries*. http://www.st.com/en/embedded-software/stm32-standard-peripheral-libraries.html (vegeu la pàgina 24).

[27] ST. *STM32 STM32Cube hardware abstraction layer (HAL)*. https://www.st.com/en/development-tools/stm32cubemx.html (vegeu la pàgina 24).

[28] Miro Samek. *Building Bare-Metal ARM Systems with GNU: Part 2*. https://www.embedded.com/design/mcus-processors-and-socs/4026075/Building-Bare-Metal-ARM-Systems-with-GNU-Part-2 (vegeu la pàgina 25).

[29] ARM. *Cortex™-M3 Technical Reference Manual - ITN*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/BABCCDFD.html (vegeu la pàgina 31).

[30] ST. *STM32F4XXX Reference Manual - RM0090*. http://www.st.com/resource/en/reference_manual/dm00031020.pdf (vegeu les pàgines 37, 48, 57, 69, 94, 105, 109, 204, 206, 231, 232).

[31] Silicon Labs. *EFM32TG840F32/F16/F8 DATASHEET* (vegeu les pàgines 41, 231).

[32] ST. *AN3371 - Using the hardware real-time clock (RTC) in STM32 F0, F2, F3, F4 and L1 series of MCUs*. https://www.st.com/resource/en/application_note/dm00025071.pdf (vegeu la pàgina 57).

[33] NXP. *PCF85063ATL*. https://www.nxp.com/part/PCF85063ATL (vegeu la pàgina 59).

[34] Maxim integrated. *DS1307*. https://www.maximintegrated.com/en/products/digital/real-time-clocks/DS1307.html (vegeu la pàgina 59).

[35] ST. *M41T82*. https://www.st.com/en/clocks-and-timers/m41t82.html (vegeu la pàgina 59).

[36] Silicon Labs. *AN0015.0: EFM32 and EZR32 WirelessMCU Series 0 Watchdog* (vegeu la pàgina 70).

[37] *putty*. https://www.putty.org/ (vegeu la pàgina 87).

[38] *Tera Term*. https://ttssh2.osdn.jp/index.html.en (vegeu la pàgina 87).

[39] *TM Terminal*. https://marketplace.eclipse.org/content/tm-terminal (vegeu la pàgina 87).

[40] Silicon Labs. *EFM32ZG108 Datasheet*. https://www.silabs.com/documents/public/data-sheets/EFM32ZG108.pdf. Silicon Labs (vegeu les pàgines 101, 206).

[41] ST. *202STM32F405xx/202STM32F407xx Datasheet*. https://www.st.com/resource/en/datasheet/dm00037051.pdf (vegeu la pàgina 101).

[42] *32-bit MCU Knowledge Base: Writing to Internal Flash*. https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2014/05/19/writing_to_internal-D0Tt (vegeu la pàgina 102).

[43] Silicon Labs. *EMLIB library - MSC*. https://siliconlabs.github.io/Gecko_SDK_Doc/efm32tg/html/group__MSC.html (vegeu la pàgina 102).

[44] Silicon Labs. *AN0003: UART Bootloader*. https://www.silabs.com/documents/public/application-notes/an0003-efm32-uart-bootloader.pdf (vegeu les pàgines 103, 232).

[45] Silicon Labs. *EFM32 Giant Gecko 11 Family Reference Manual*. https://www.silabs.com/documents/public/reference-manuals/efm32gg11-rm.pdf. Silicon Labs (vegeu la pàgina 109).

[46] Wikipedia contributors. *Media-independent interface — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Media-independent_interface&oldid=851943072. [Online; accessed 8-August-2018]. 2018 (vegeu la pàgina 109).

[47] Viquipèdia. *Controller area network — Viquipèdia, l'Enciclopèdia Lliure*. [Internet; descarregat 21-maig-2018]. 2018. URL: %5Curl%7B//ca.wikipedia.org/w/index.php?title=Controller_area_network&oldid=19926922%7D (vegeu la pàgina 109).

[48] Inc. Free Software Foundation. *LwIP - A Lightweight TCP/IP stack*. LwIP-ALightweightTCP/IPstack (vegeu la pàgina 110).

[49] Wikipedia contributors. *LwIP — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=LwIP&oldid=839596879. [Online; accessed 8-August-2018]. 2018 (vegeu la pàgina 110).

[50] Silicon Labs. *AN0025: Peripheral Reflex System*. https://www.silabs.com/documents/public/application-notes/an0025_efm32_prs.pdf (vegeu la pàgina 110).

[51] FreeRTOS. *FreeRTOS Homepage*. http://www.freertos.org (vegeu la pàgina 127).

[52] Richard Barry. *Mastering the FreeRTOS™ Real Time Kernel*. https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf (vegeu les pàgines 128, 145, 148, 150, 165, 255).

[55] FreeRTOS. *Running the RTOS on a ARM Cortex-M Core*. https://www.freertos.org/RTOS-Cortex-M3-M4.html (vegeu la pàgina 159).

[56] Viquipèdia. *Autòmat finit — Viquipèdia, l'Enciclopèdia Lliure*. [Internet; descarregat 23-febrer-2018]. 2018. URL: %5Curl%7B//ca.wikipedia.org/w/index.php?title=Aut%C3%B2mat_finit&oldid=19686277%7D (vegeu la pàgina 180).

[57] Wikipedia contributors. *Extended finite-state machine — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Extended_finite-state_machine&oldid=711433361. [Online; accessed 2-January-2019]. 2016 (vegeu la pàgina 181).

[58] Silicon Labs. *EMDRV*. https://siliconlabs.github.io/Gecko_SDK_Doc/efm32tg/html/group__emdrv.html (vegeu la pàgina 192).

[59] Silicon Labs. *RTCDRV*. https://siliconlabs.github.io/Gecko_SDK_Doc/efm32tg/html/group__RTCDRV.html (vegeu la pàgina 192).

[60] Silicon Labs. *SLEEP*. https://siliconlabs.github.io/Gecko_SDK_Doc/efm32g/html/group__SLEEP.html (vegeu la pàgina 192).

[62] Keil. *Using Cortex-M3/M4/M7 Fault Exceptions*. http://www.keil.com/appnotes/files/apnt209.pdf (vegeu la pàgina 199).

[63] Niall Cooling. *Developing a Generic Hard Fault handler for ARM Cortex-M3/Cortex-M4*. https://blog.feabhas.com/2013/02/developing-a-generic-hard-fault-handler-for-arm-cortex-m3cortex-m4/ (vegeu la pàgina 199).

[64] ARM. *Cortex-M3 Devices Generic User Guide - HardFault Status Register*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/Cihdjcfc.html (vegeu la pàgina 199).

[65] *Debug a HardFault*. https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2014/05/26/debug_a_hardfault-78gc (vegeu la pàgina 199).

[66] ST. *Description of STM32F4 HAL and LL drivers - UM1725*. https://www.st.com/resource/en/user_manual/dm00105879.pdf (vegeu la pàgina 204).

[67] ST. *STM32L011x3 STM32L011x4 Datasheet*. https://www.st.com/resource/en/datasheet/stm32l011d4.pdf. ST (vegeu la pàgina 206).

[68] ST. *AN4865 - Low-power timer (LPTIM) applicative use-cases on STM32 MCUs*. https://www.st.com/resource/en/application_note/dm00290631.pdf (vegeu la pàgina 208).

[69] NXP. *LPTMR - Low Power Timer*. https://community.nxp.com/docs/DOC-99954 (vegeu la pàgina 208).

[70] FreeRTOS. *FreeRTOS Homepage - Low Power Support*. https://www.freertos.org/low-power-tickless-rtos.html (vegeu la pàgina 209).

[71] Doxygen. *Doxygen documention*. http://www.stack.nl/~dimitri/doxygen/manual/docblocks.html (vegeu la pàgina 211).

[72] GitHub. *GitHub pages*. https://pages.github.com/ (vegeu la pàgina 212).

[73] ARM Ltd. *CMSIS-CORE support for Cortex-M processor-based devices*. http://arm-software.github.io/CMSIS_5/Core/html/index.html (vegeu la pàgina 215).

[74] ARM Ltd. *CMSIS DSP Software Library*. http://arm-software.github.io/CMSIS_5/DSP/html/index.html (vegeu la pàgina 216).

[75] Silicon Labs. *AN0051: Digital Signal Processing with the EFM32*. https://www.silabs.com/documents/public/application-notes/AN0051.pdf (vegeu les pàgines 216, 221).

[76] ARM Ltd. *CMSIS-RTOS2 Real-Time Operating System: API and RTX Reference Implementation*. http://arm-software.github.io/CMSIS_5/RTOS2/html/index.html (vegeu la pàgina 216).

[77] ST. *Developing Applications on STM32Cube with RTOS - UM1722*. https://www.st.com/resource/en/user_manual/dm00105262.pdf (vegeu la pàgina 216).

[78] Keil. *Keil RTX*. http://www.keil.com/pack/doc/CMSIS/RTOS2/html/rtx5_impl.html (vegeu la pàgina 216).

[79] ARM Ltd. *CMSIS-DAP Interface Firmware for CoreSight Debug Access Port*. http://arm-software.github.io/CMSIS_5/DAP/html/index.html (vegeu la pàgina 216).

[80] Vikas Chandra Liangzhen Lai Naveen Suda. *CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs*. https://arxiv.org/abs/1801.06601 (vegeu la pàgina 216).

[81] ARM Ltd. *CMSIS NN Software Library*. http://arm-software.github.io/CMSIS_5/NN/html/index.html (vegeu la pàgina 216).

[82] Gerard Holzmann. *The Power of 10: Rules for Developing Safety-Critical Code*. https://www.researchgate.net/publication/220477862_The_Power_of_10_Rules_for_Developing_Safety-Critical_Code. 2006. DOI: DOI:10.1109/MC.2006.212 (vegeu la pàgina 217).

[83] Jet Propulsion Laboratory - California Institute of Technology. *JPL Institutional Coding Standard for the C Programming Language*. https://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf. 2009 (vegeu les pàgines 217, 219).

[84] HORIBA MIRA Ltd. *MISRA C*. https://www.misra.org.uk/Activities/MISRAC/tabid/171/Default.aspx (vegeu les pàgines 217, 218).

[85] Michael Barr. *Embedded C Coding Starndard*. https://barrgroup.com/Embedded-Systems/Books/Embedded-C-Coding-Standard. 2013 (vegeu les pàgines 217, 218).

[88] HORIBA MIRA Ltd. *MISRA C: 2012 Amendment 1*. https://misra.org.uk (vegeu la pàgina 218).

[89] Jet Propulsion Laboratory. *Laboratory for Reliable Software (LaRS)*. https://lars-lab.jpl.nasa.gov/ (vegeu la pàgina 219).

[90] Barr Group. *How to Get Started with C++ in Embedded Systems*. https://barrgroup.com/Embedded-Systems/How-To/Getting-Started-With-Cplusplus (vegeu la pàgina 223).

[91] Alan Dorfmeyer i Pat Baird. *Interrupts in C++*. https://www.embedded.com/design/prototyping-and-development/4023817/Interrupts-in-C- (vegeu la pàgina 227).

[92] Bill Gatliff. *Implementing Interrupt Service Routines in C++*. http://www.drdobbs.com/implementing-interrupt-service-routines/184401485?pgno=3 (vegeu la pàgina 227).

[93] Silicon Labs. *User Manual Start Kit EFM32TG-STK3300* (vegeu la pàgina 232).

[94] Silicon Labs. *EFM32TG Data Sheet*. https://www.silabs.com/documents/public/data-sheets/efm32tg-datasheet.pdf. Silicon Labs (vegeu les pàgines 233, 234).

[95] Silicon Labs. *EFM32ZG Datasheet*. https://www.silabs.com/documents/public/data-sheets/efm32zg-datasheet.pdf. Silicon Labs (vegeu la pàgina 233).

[96] Silicon Labs. *EFM32HG Datasheet*. https://www.silabs.com/documents/public/data-sheets/efm32hg-datasheet.pdf. Silicon Labs (vegeu la pàgina 233).

[97] Viquipèdia. *IEEE 754 — Viquipèdia, l'enciclopèdia lliure*. [Internet; decarregat 11-July-2020]. 2020. URL: //ca.wikipedia.org/w/index.php?title=IEEE_754&oldid=24054533 (vegeu la pàgina 239).

[98] ARM. *ARM Cortex-M7 Processor Technical Reference Manual*. https://static.docs.arm.com/ddi0489/f/DDI0489F_cortex_m7_trm.pdf (vegeu la pàgina 242).

[100] Michael Barr. *Introduction to Rate Monotonic Scheduling*. https://barrgroup.com/Embedded-Systems/How-To/RMA-Rate-Monotonic-Algorithm (vegeu la pàgina 257).

[102] A. Gosavi. *Queuing Formulas*. http://web.mst.edu/~gosavia/queuing_formulas.pdf (vegeu la pàgina 259).

[103] Ivo Adan. *The M/M/1 system*. http://www.win.tue.nl/~iadan/que/h4.pdf (vegeu la pàgina 259).

[104] Wikipedia contributors. *M/M/1 queue — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=M/M/1_queue&oldid=819613747. [Online; accessed 27-July-2018]. 2018 (vegeu la pàgina 259).

[105] David Kalinsky. *How to size message queues*. https://www.embedded.com/design/other/4024545/How-to-size-message-queues (vegeu la pàgina 259).

[106] Jack Ganssle. *My favorite software debouncers*. https://www.embedded.com/electronics-blogs/break-points/4024981/My-favorite-software-debouncers (vegeu la pàgina 263).

[107] David B. Stewart. *How to Choose A Sensible Sampling Rate*. https://www.embedded.com/design/configurable-systems/4006414/How-to-Choose-A-Sensible-Sampling-Rate (vegeu la pàgina 263).

Llibres

[3] Jon Wilson, editor. *Sensor Technology Handbook*. Elsevier Inc., 2004. ISBN: 9780750677295 (vegeu la pàgina 14).

[17] K.C. Wang. *Embedded and Real-Time Operating Systems*. Springer, 2017. ISBN: 9783319515168. DOI: https://doi.org/10.1007/978-3-319-51517-5 (vegeu la pàgina 20).

[22] Trevor Martin. *The Designer's Guide to the Cortex-M Processor Family*. Volum Second edition. Newnes, 2016. ISBN: 9780081006290 (vegeu les pàgines 20, 199, 221).

[23] Joseph Yiu. *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Volum Third edition. Newnes, 2013. ISBN: 9780124080829 (vegeu les pàgines 22, 38, 221).

[53] Jack Ganssle i Michael Barr. *Embedded Systems Dictionary*. CMPBooks, 2003. ISBN: 1578201209 (vegeu la pàgina 143).

[54] Michael Barr i Anthony Massa. *Programming Embedded Systems*. second edition. O'Reilly, 2007. ISBN: 980596009830 (vegeu la pàgina 143).

[61] *ARM System Developer's Guide*. The Morgan Kaufmann Series in Computer Architecture and Design. Burlington: Morgan Kaufmann, 2004. ISBN: 978-1-55860-874-0. DOI: https://doi.org/10.1016/B978-155860874-0/50022-8 (vegeu la pàgina 199).

[86] *C and C++ Coding Standards*. ESA Board for Software Standardisation i Control, 2001 (vegeu la pàgina 217).

[87] *MISRA C:2012*. MIRA Limited, març de 2013. ISBN: 9781906400118 (vegeu la pàgina 218).

[99] Jiacung Wang. *Real-Time Embedded Systems*. Wiley, 2017. ISBN: 9781118116173 (vegeu la pàgina 255).

[101] Peter Marwedel. *Embedded System Design*. Springer, 2006. ISBN: 9781402076909 (vegeu la pàgina 257).

Aquesta pàgina està en blanc expressament, tot va bé.

Glossari

ARM
: Empresa anglesa que dissenya i comercialitza arquitectures de processadors i eines associades 19, 25, 221

Buffer circular
: Estructura de dades que utilitza un sol buffer i que es pot accedir-ho de forma circular 86–88, 98, 99, 159

Callback
: Funció que es crida en quan acaba un procés 96, 98, 192

CMSIS
: *Cortex Microcontroller Software Interface Standard 215*

Cortex
: *Arquitectura de microcontroladors de la companyia ARM 19, 20*

CP2102
: *Dispositiu conversor de USB a sèrie, força utilitzat per comunicar microcontroladors amb ordinadors a través del port USB 85, 86*

CPU
: Central Processor Unit *Unitat central de procès, part que fa tot el còmput i maneig de dades dins un processador 19, 54*

Dead-lock
: *Situació en que dues o més tasques estan bloquejades esperant-se una a l'altra 69*

Duty cycle
: *Cicle de treball, és la relació que existeix entre el temp que un senyal esta a '1' i el periode d'aquest senyal 61–63, 118, 120, 122, 163*

Flag
: *Un* flag *és un bit que indica algun valor determinat d'un dispositiu o dada 48, 50, 53, 80*

FLASH
: *Memòria no-volàtil de gran capacitat i baix consum 14, 34, 93, 101, 102, 223, 232, 267*

FreeRTOS
: *Sistema Operatiu de Temps Real de codi obert i lliure 127, 130, 163*

FW
: Firmware*, Aquell software guardat i executat en un microcontrolador 41, 69, 231–233*

GCC
: GNU C Compiler], Compilador de C de GNU 25

Layout
: Dibuix final de com queda la PCB a dissenyar 232

Macro
: Tros de codi que té assignat un nom, de manera que cada cop que s'inserta el nom es canvia pel codi definit prèviament 267

Memory mapped
: Assignar una posició de memòria a un component HW per facilitar el seu accés 22

Open drain
: Tipus de sortida que només pot forçar un valor de '0', per tant li cal un pull-up per funcionar correctament 121

PCB
: *Printed Circuit Board* Placa de circuit imprès on es solden els components Hardware 14, 16, 90, 224, 231, 233

PLC
: Programmable logic controller, autòmats industrials 20

Pull-down
: Resistència connectada a terra per forçar un valor d''0' a una línia o bus 42

Pull-up
: Resistència connectada a alimentació per forçar un valor d''1' a una línia o bus 41, 90, 121, 288

PWM
: Pulse Width Modulation 52, 61–63, 117, 118, 120, 122, 163, 208

Race condition
: Error provocat per l'exeucuó simultània de dues o més tasques 145, 149

Stack
: Regió de memòria on emmagatzemar dades pròpies d'una tasca 130, 137, 199

Systick
: *Timer* integrat dins els *cores* Cortex-M 38

Task
: Funció que implementa un procés de SO juntament amb l'*stack* corresponent 127

Tick
: Esdeveniment periòdic per a que el Sistema Operatiu prengui el controll de l'execució 131

Timer
: Comptador segons un rellotge que genera una interrupció quan arriba a un cert valor configurable 51, 57, 61

Watchdog
: Perifèric que reinicia el sistema si no s'hi accdeix periòdicament 69

Wrapper
: Tros de codi que embolcalla un altre per donar-li una interfície comuna 216

Aquesta pàgina està en blanc expressament, tot va bé.

Acrònims

ADC
: Analog to Digital Converter 14, 75–77, 79, 89, 93, 127, 183, 206, 296

API
: *Application programming interface 215*

ASIC
: *Application Specific Integrated Circtuit (circuit integrat d'aplicació específica) 19*

bps
: *Bits per segon 85*

BSP
: Board Support Package 43, 232

CAN
: *Controller area network 109, 215*

DAC
: *Digital to Analog Converter 79, 82, 89*

DMA
: Direct Memory Access 95, 96, 98, 109

DSP
: Digital Signal Processor (Processador Digital de Senyal) 19, 216, 221

EFSM
: *Extended finite state machine 181*

ESA
: European Space Agency 217

FPGA
: Field Programmable Gate Array 19

FSM
Finite state machine 180, 195

GPIO
General Purpouse Input/Output 22, 31, 41, 42, 47, 225, 264

I2C
Inter-Integrated Circuit 58, 59, 89, 90, 92, 93, 118, 150, 165

IDE
Integrated development environment, *Entorn integrat de desenvolupament 16, 31*

IRQ
Interrupt request, *Petició d'interrupció 47, 48, 50, 53, 75, 87, 88, 98, 99, 110, 121, 128, 132, 206, 207, 209, 231*

ISR
Interrupt Service Routine, *Rutina de servei d'interrupció 47, 48, 50, 53, 58, 62, 70, 80, 82, 86–88, 99, 101, 110, 121, 122, 128, 132, 135, 144–146, 153, 154, 159, 178, 199, 227, 264, 301*

LCD
Liquid Crystal Display *Pantalla de Cristall Líquid 109*

LED
Light Emission Diode 14, 53, 122, 146, 224

MAC
Multiply and Accumulate *Instrucció de multiplicar i acumular 221*

MCI
Memory Card Interface 216

MCU
MicroController Unit 16

MII
Media-independent interface 109

MISO
Master Input Slave Output, Entrada al Master Sortida de l'Esclau 93

MOSI
Master Output Slave Input, Sortida del Master Entrada a l'Esclau 93

NXP
NXP Semiconductors 20

OS
Operating System, Sistema Operatiu 127, 143

pdFALSE
Valor lògic "Fals" definit a FreeRTOS 146, 149

pdTRUE
Valor lògic "Cert" definit a FreeRTOS 146, 149

PRS
Peripheral Reflex System 110

RAM
Random Access Memory 14, 16, 19, 232

ROM
: *Read-Only Memory 16, 19, 101*

RTC
: *Real Time Clock 52, 58*

RTOS
: Real-Time Operating System*, Sistema Operatiu de Temps Real 29, 127, 130, 163, 166, 205, 216, 259*

RTTI
: Run-time type information *Informació de tipus en temps d'execució 224*

SAI
: Serial Audio Interface 216

SCL
: Serial Clock Line (I2C) 89, 90

SCLK
: *Serial Clock*, Relloge Master 93

SDA
: Serial Data (I2C) 89, 90

SDIO
: *Secure Digital Input Output 109*

SiLabs
: *Silicon Labs 20*

SIMD
: Single Instruction, Multiple Data *Única instrucció, Múltiples dades 221*

SPI
: *Serial Peripheral Interface 89, 93*

SS
: Slave Select*, Selecció d'Esclau 93*

ST
: *STMicroelectronics 20*

TI
: *Texas Instruments 20*

UART
: Universal Asynchronous Receiver-Transmitter 85, 227

USART
: *Universal Synchronous and Synchronous Receiver-Transmitter 85, 86*

USB
: *Universal Serial Bus 85, 86*

Vdd
: *Voltatge d'alimentació 76, 82*

Aquesta pàgina està en blanc expressament, tot va bé.

Índex de figures

1.1 Fotografia de la placa de desenvolupament de SiliconLabs 15
1.2 Cables dupont 15
1.3 Aspecte del IDE Simplicity Studio™de SiliconLabs 17

2.1 Raspberry Pi amb un processador Cortex-A 21
2.2 Diferents microcontrolador Cortex-M0 i M3 21
2.3 Placa Freescale FRDM-KL25Z amb un Cortex-M0+ 21
2.4 Mapa de memòria d'un Cortex-M3 23
2.5 Registres de la DI a usar a l'exemple (24, pàgina 24) 23

3.1 Captura de pantalla de la Consola del Simplicity Studio 32

4.1 Opció a *Simplicity* per activar el punt flotant al **printf()** 35

6.1 Esquemàtic mostrant un LED connectat a un pin de GPIO 42
6.2 Esquemàtic amb resistències de *pull-ups* 42

7.1 Vectors d'interrupció 49

8.1 Diagrama de seqüència de l'exemple Timer_2 55

9.1 Registres del RTC de STM32 (32) 57

10.1 Generació de PWM amb un timer (24, pàgina 262) 62
10.2 PWM amb Duty Cycle al 16% 63
10.3 PWM amb Duty Cycle al 50% 64
10.4 PWM amb Duty Cycle al 83.3% 64
10.5 PWM amb Duty Cycle al 100% 65
10.6 Fotografia del servomotor Parallax usat 66
10.7 PWM per situar el servomotor a 0 ° 67
10.8 PWM per situar el servomotor a 180 ° 68

11.1 Funcionament del em Watchdog (36) 70

12.1 Esquemàtic de la connexió del Potenciòmetre al canal d'ADC 77
12.2 Fotografia del sistema amb el connexitat correcte 77

13.1 Senyal capturat per l'oscil·loscopi. 83
13.2 Detall del senyal capturat per l'oscil·loscopi 84

14.1 Connexió del CP2102 a la placa de prototipat 87

15.1 Esquema d'un bus I2C típic 90

16.1 Bus SPI típic 94

19.1 Consola de l'exemple AES_1 107
19.2 Web per desxifrar el text xifrat de l'exemple AES_1 107

22.1 Estats possibles d'una tasca 129
22.2 Diagrama de seqüència de dues tasques 133
22.3 Diagrama de seqüència de dues tasques correcte 134

24.1 Consola amb la sortida de l'exemple FreeRTOS_Delay 141
24.2 Captura de l'oscil·loscopi de l'exemple vTaskDelayUntil() 142

25.1 Diagrama de seqüència de l'exemple amb semàfors 144
25.2 Diagrama de seqüència de l'exemple de cues 146
25.3 Temps de resposta usant semàfors 156
25.4 Temps de resposta usant notificació a tasca 156
25.5 Temps de resposta usant groups de notificació 157

31.1 FSM per l'exemple GPIO_1 180
31.2 FSM d'un termòstat senzill 183

33.1 Debugger aturat a la instrucció DEBUG_BREAK i el *dump* els registres 201
33.2 Codi assemblador a la posició de memòria indicada pel registre **PC** 202

34.1 *Shadow registers* del perifèric RTC dels STM32 (30, pàgina 800) 204

35.1 Captura de les mesures de temps amb l'analitzador lògic 207

36.1 Comentari per doxygen dins un codi 212
36.2 Botons de Simplicity, l'arroba blava permet executar Doxygen 212
36.3 Configuració de Doxygen dins de Simplicity 213
36.4 Pàgina web de documentació vista dins de Simplicity Studio 213

39.1 Configuració del Simplicity Studio afegint-hi la biblioteca CMSIS-DSP 222

40.1 Configuració Simplicity Studio per deshabilitar RTTI i les excepcions 224

41.1 Pinout pels microcontroladors 234

42.1 Reset Handler per Cortex-M 235
42.2 Configuració del *Debugger* 236
42.3 Còpia de la secció .bss a la memòria RAM 237
42.4 Inicialització del registre d'stack (Stack Pointer) 237
42.5 Inicialització de la secció .bss 237

42.6 Crida a la funció _init() i funció main() 238

51.1 Diagrama de seqüència de l'exemple de *debounce* 265

53.1 Disposició de l'estructura a la memòria 270
53.2 Disposició de l'estructura empaquetada a la memòria 271
53.3 Detall de la finestra de *memory dump* a Simplicity Studio 273

Aquesta pàgina està en blanc expressament, tot va bé.

Índex de llistats

2.1 Accedint a memòria en C 23
2.2 Exemple de definició d'estructura per accedir a memòria 24
2.3 Declaració d'una variable d'accés a la memòria estructurada 24
2.4 Ús de l'estructura d'accés 24
2.5 Codi velocitat d'un microcontrolador 26
4.1 Funció **_write()** 33
4.2 Redefenir **printf()** 34
5.1 Exemple de configuració del rellotge pel RTC 38
5.2 Configuració del *Systick* 38
5.3 ISR del *Systick* 39
5.4 Funció delay() amb *Systick* 39
6.1 Codi d'exemple de GPIO 43
6.2 Codi de configuració d'un pin 43
6.3 Codi amb la nova configuració del pin 44
6.4 Exemple de BSP senzill 44
6.5 Manipulant un bit concret d'una variable 45
7.1 Exemple d'ISR per GPIO 48
7.2 Exemple d'ISR per AVR 50
7.3 Exemple d'ISR per AVR 50
8.1 Codi d'exemple d'ús d'un Timer 52
8.2 Codi per comprovar si el Timer ha arribat a cert valor 53
8.3 Codi corresponent a l'activació de les IRQs del Timer 53
8.4 ISR del Timer 54
8.5 ISR del GPIO per l'exemple del Timer 54
9.1 Inicialització del RTC 58
9.2 ISR del RTC 58
10.1 Configuració del Timer per l'exemple PWM 62
10.2 Configuració del Timer per l'exemple PWM 62

10.3 Funció que calcula els *counts* donat els graus que es vol del servomotor 67
10.4 ISR del botó que incrementa la rotació del servomotor 68
11.1 ISR del botó que alimenta el *Watchdog* 70
11.2 Codi per detectar la causa del reinici 71
12.1 Codi de lectura de l'ADC 76
13.1 Inicialització del DAC 80
13.2 Bucle infinit del DAC 80
13.3 Part de la ISR del Timer per generar la dada pel DAC 83
14.1 ISRs de TX i RX de la UART 86
14.2 Exemple ISR avançada 88
14.3 Exemple ISR avançada 88
14.4 Funció main 88
15.1 Initialització dels pins per l'I2C 91
15.2 Inicialització del perifèric I2C 91
15.3 Funció **I2C_ReadRegister** 91
15.4 Funció **testI2C()** 92
17.1 Inicialització del DMA 96
17.2 Definició de la variable **dmaControlBlock** 96
17.3 Configuració del canal DMA 96
17.4 Callback del DMA **DmaComplete()** 97
17.5 Paràmetres de configuració del DMA 97
17.6 Activació de la transferència DMA 97
17.7 Comparació dels dos buffers de l'exemple DMA 98
17.8 Diferències a la inicialització del DMA 98
17.9 Funció sendUARTbyDMA() 99
17.10Funció **main()** de l'exemple 99
18.1 Estructura per guardar-se a la FLASH 102
18.2 Funcions per accedir a la FLASH 102
19.1 Clau i text a xifrar 106
19.2 Operació de xifratge 106
19.3 Operació de xifratge 106
20.1 Configuració del Timer i l'*input capture* 111
20.2 Configuració del GPIO per generar un senyal PRS 111
20.3 ISR del Timer 112
20.4 Configuració de l'ADC perquè funcioni amb el PRS 112
20.5 Configuració del Timer0 perquè funcioni amb el PRS 113
20.6 Configuració del DMA per obtenir dades de l'ADC 113
20.7 Configuració del DMA per obtenir dades de l'ADC 114
20.8 Configuració del DAC perquè funcioni amb el PRS 114
20.9 Configuració del PRS i el Timer 115
20.10Configuració del PRS i el Timer 115
20.11ISR del botó 1 116
21.1 Part de la funció **I2C_WriteRegister** 118
21.2 Funció PWM_Set() 118
21.3 Funció **APDS_9960_InitProximity()** 119
21.4 Funció **APDS_9960_ReadProximity()** 119
21.5 Funció principal 120
21.6 Nova funció d'initialització del APDS_9960 122
21.7 Habilitar l'interrupció del pin corresponent 122

21.8 ISR amb el *flag* . . . 122
21.9 Funció principal amb suport d'interrupcions . . . 123
22.1 Esquelet d'una tasca . . . 130
22.2 Codi ISR d'exemple . . . 132
23.1 Tasca TaskLedToggle per FreeRTOS . . . 137
23.2 Main HelloWorld per FreeRTOS . . . 138
24.1 Tasca de l'exemple FreeRTOS_BlinkTask . . . 139
24.2 Tasca de l'exemple FreeRTOS_Delay . . . 142
25.1 Tasca amb semàfor d'exemple . . . 144
25.2 ISR del botó 0 . . . 145
25.3 Part del codi d'una de les ISRs . . . 146
25.4 Part principal de la tasca TaskLedToggle . . . 147
25.5 Creació d'una cua . . . 147
25.6 Paquet dins d'estructura . . . 147
25.7 Creació de la cua amb un paquet de dades . . . 148
25.8 Rebre un paquet de dades de la cua . . . 148
25.9 Exemple d'ús de macros en C . . . 149
25.10 Sortida de la consola sense Mutex . . . 149
25.11 Sortida de la consola amb Mutex . . . 149
25.12 Tasca esperant per un grup d'esdeveniments . . . 151
25.13 ISR notificant un esdeveniment a un grup . . . 151
25.14 Tasca esperant un grup d'esdeveniments (OR) . . . 152
25.15 Creació del conjunt de cues . . . 153
25.16 Tasca que fa servir el conjunt de cues . . . 153
25.17 Tasca que espera la notificació . . . 154
25.18 Tasca que espera la notificació . . . 155
26.1 ISR de RX de la UART amb FreeRTOS . . . 159
26.2 ISR de TX de la UART amb FreeRTOS . . . 160
26.3 funció UART_Send() per FreeRTOS . . . 160
26.4 Tasca principal de l'exemple . . . 160
27.1 Tasca ReadSensor . . . 164
27.2 Tasca MngData . . . 164
27.3 Creació de tasques . . . 164
27.4 Part de la funció I2C_WriteRegister() adaptada a FreeRTOS . . . 165
28.1 Codi d'exemple de la tasca de control del **watchdog** . . . 168
29.1 Inicialització del *wrapper* I2C amb Mutex . . . 170
29.2 Modificacions a les funcions *wrapper* I2C amb Mutex . . . 171
29.3 Afegint més dades a l 'estructura del *wrapper* I2C amb mutex . . . 172
31.1 funció main() d'Arduino . . . 180
31.2 Codi d'exemple de GPIO . . . 181
31.3 Codi d'exemple de GPIO . . . 182
31.4 funció ADCGetValue() . . . 184
31.5 funció getTemperature() . . . 184
31.6 Codi de la FSM per un termòstat senzill . . . 185
31.7 Estructura bàsica d'una FSM . . . 186
31.8 Codi de termòstat amb l'estructura bàsica d'una FSM . . . 187
31.9 Funció de loop amb múltiples FSMs . . . 188
31.10 Estructura bàsica d'un *kernel* per múltiples FSMs . . . 189
32.1 FSM amb control del temps . . . 192

32.2 Estructura bàsica de les tasques programades 193
32.3 Estructura bàsica de la funció Executa_kernel() 193
32.4 Estructura bàsica de la funció Executa_tasca() 194
33.1 Codi HardFault_Handler 200
33.2 Codi HardFault_Handler (continuació) 200
35.1 Bucle principal amb funcions de baix consum 207
35.2 Exemple ús de **LETIMER** 208
40.1 Part del codi de la classe LED 225
40.2 Part del codi de la classe LED 226
40.3 Ús de l'operador << de la classe UART 227
40.4 Implementació de l'operador << per la classe UART 228
40.5 Implementació de les ISRs en C++ 228
40.6 Part del fitxer UART.cpp de l'exemple d'us del *driver* en C++ per la UART 229
43.1 funció mulf() 239
43.2 funció muld() 240
43.3 codi assemblador de la funció mulf.c 240
43.4 codi assemblador de la funció mulf.c usant FPU 241
43.5 codi assemblador de la funció muld.c 241
43.6 codi assemblador de la funció muld.c usant FPU de Cortex-M7 242
51.1 ISR del timer per fer debouncing 264
51.2 Codi per engegar el timer a la ISR del GPIO 264
52.1 Diferents implementacions de **printf()** 268
53.1 Estructura d'exemple 269
53.2 Estructura d'exemple empaquetada 270
53.3 Mostrant una estructura *byte* a *byte* 272

Index de funcions

Symbols

& 44
^ 44
__eabi_dmul() 240
__eabi_fmul() 240
__libc_init_array() 236
_start() 25, 236
_write() 33
~ 44

A

ADC_DataSingleGet() 76, 183, 207
ADC_InitSingle() 112
ADC_Start() 76, 183, 207
ADCGetValue() 183
AES_DecryptKey128() 106
AES_ECB128() 106
any_IRQHandler() 132
APDS_9960_InitProximity() 118, 163
APDS_9960_InitProximity_IRQ() 121
APDS_9960_ReadProximity() . 118, 120, 122, 163
AvailableData() 87, 88, 99

B

BSP_Init() 117, 144

C

calc_values() 186
check_buffers_copy() 97
CircularBuffer class 227
CMU_ClockDivSet() 37, 58, 208
CMU_ClockEnable() 37, 42, 79, 90
CMU_ClockFreqGet() 38
CMU_ClockSelectSet() 37, 58, 208

D

DAC_ChannelOutputSet() 80, 82
DAC_Init() 79
DAC_InitChannel() 79, 112
DAC_PrescaleCalc() 79
DEBUG_BREAK 202
degrees_to_pwm() 66
Delay() 38
DMA_ActivateAuto() 96
DMA_ActivateBasic() 98, 112, 115
DMA_CfgChannel() 96, 112, 115
DMA_CfgDescr() 96, 112, 115
DMA_ChannelEnabled() 98
DMA_Init() 96
DmaComplete() 97

E

EMU_EnterEM1() 110, 207

EMU_EnterEM3() 208

F

Flash_Read() 102
Flash_Write() 102
fp() 202
free() 247

G

getTemperature() 183
GPIO_EVEN_IRQHandler()48, 54, 66, 70, 79, 122, 144, 264
GPIO_InputSenseSet() 110
GPIO_IntClear() 48, 54, 70, 122
GPIO_IntConfig() 47, 110, 121
GPIO_IntGet() 48, 54, 70, 122
GPIO_ODD_IRQHandler() 79
GPIO_PinInGet() 52, 264
GPIO_PinModeSet() 43, 90, 110, 121
GPIO_PinOutClear() 43, 48
GPIO_PinOutSet() 43
GPIO_PinOutToggle() 52, 58, 208

H

HardFault_Handler() 199

I

I2C_Init() 90
I2C_initialize 165
I2C_initialize() 171
I2C_ReadRegister() 90, 92, 118, 165
I2C_Transfer() 90, 92, 118, 165
I2C_TransferInit() 90, 92, 118, 165
I2C_WriteRegister() 118, 121, 165
ITM_SendChar() 31, 33

L

LED::Off() 224
LED::On() 224
LED::Toggle() 224
LedInit() 43
LedOff() 43
LedOn() 43
LedToggle() 137, 139, 144, 146
LETIMER0_IRQHandler() 208
LETIMER_CompareSet() 208
LETIMER_IntClear() 208
LETIMER_IntGet() 208
loop() 186

M

main() 25, 26, 29, 44, 52, 54, 80, 99, 120, 122, 137, 144, 146, 159, 163, 186, 192, 207, 208, 227
malloc() 247
memcmp() 218
MngData_task() 163
MSC_ErasePage() 102
MSC_WriteWord() 102
muld() 239
mulf() 239
my_DMA_Init() 98
my_HardFault_Handler() 199

N

next_estate() 186
NVIC_ClearPendingIRQ() 58
NVIC_EnableIRQ() 47, 53, 58, 110, 121, 215

O

OneTask() 130

P

pdMS_TO_TICKS() 137, 139
PopData() 87, 99
portYIELD_FROM_ISR() 132, 144, 146, 150, 159
PRINTF() 34
printf() 33, 34, 80, 163, 267
printf_char() 267
printf_hex8() 267
printf_int() 267
printf_string() 267
PRS_SourceSignalSet() 110, 112, 115
PushData() 87
PWM_Init() 118
PWM_Set() 118, 120, 163

R

read_inputs() 186
ReadSensor_task() 163
Registra_tasca() 192

Reset_Handler . . . 235
ResetHandler . . . 235
RTC_CompareSet() . . . 58
RTC_IntClear() . . . 58
RTC_IntEnable() . . . 58
RTC_IRQHandler() . . . 58
RTCDRV_AllocateTimer() . . . 194
RTCDRV_StartTimer() . . . 192, 194

S

SemaphoreHandle_t . . . 167
sendUARTbyDMA() . . . 98, 99
setup() . . . 186
setupSWOForPrint() . . . 31, 33, 117
sprintf() . . . 99
SystemInit() . . . 25, 235
SysTick_Config() . . . 38
SysTick_Handler() . . . 38

T

TaskLedToggle() . . . 137, 139, 144, 146
taskYIELD() . . . 149
testI2C() . . . 92
TIMER0_IRQHandler() . . . 53, 82, 110
TIMER1_IRQHandler() . . . 264
TIMER_CaptureGet() . . . 110
TIMER_CompareBufSet() . . . 62, 118
TIMER_CompareGet() . . . 51
TIMER_CompareSet() . . . 51
TIMER_CounterGet() . . . 52
TIMER_CounterSet() . . . 52, 54, 112, 115
TIMER_Enable() . . . 51, 52, 54, 112, 115, 264
TIMER_Init() . . . 110, 112, 115
TIMER_InitCC() . . . 62, 110
TIMER_IntClear() . . . 53, 110, 264
TIMER_IntEnable() . . . 53, 110
TIMER_IntGet() . . . 53, 110, 264
TIMER_TopBufSet() . . . 112, 115
TIMER_TopGet() . . . 51
TIMER_TopSet() . . . 51, 54, 62, 264

U

UART class . . . 227
UART::<< . . . 227
UART::AvailableData() . . . 227
UART::GetData() . . . 227
UART::SendData() . . . 227
UART::Tx() . . . 227
UART::USART1_RX_IRQHandler() . . . 227
UART::USART1_TX_IRQHandler() . . . 227
UARTTask() . . . 159
ulTaskNotifyTake() . . . 154
ulTaskNotifyTake()() . . . 154
USART1_RX_IRQHandler() . . . 88, 159, 227
USART1_TX_IRQHandler() 86, 88, 159, 227
USART_IntClear() . . . 86, 88, 159
USART_Rx() . . . 86, 88, 159
USART_Send() . . . 86, 88, 159
USART_Tx() . . . 85, 159

V

vApplicationStackOverflowHook() . . . 131
vPortSetupTimerInterrupt() . . . 209
vTaskDelay() . . . 131, 137, 139, 140, 146, 163, 167, 209
vTaskDelayUntil() . . . 131, 139, 140
vTaskNotifyGiveFromISR() . . . 154
vTaskStartScheduler() . . . 137, 138

W

watchdogClear() . . . 167
watchdogTask() . . . 167
watchdogTouch() . . . 167
WDOG_Feed() . . . 69, 70, 167
wrapper_I2C_Init() . . . 169
wrapper_I2C_ReadReg() . . . 171
wrapper_I2C_WriteReg() . . . 171
write_outputs() . . . 186
WrongfunctionAlign() . . . 202
WrongfunctionDiv0() . . . 202
WrongfunctionWrongMemory() . . . 202

X

xEventGroupSetBitsFromISR() . . . 150
xEventGroupWaitBits() . . . 150
xQueueCreate() . . . 146, 147, 152
xQueueCreateSet() . . . 152
xQueueReceive() . . . 146, 148, 159, 163
xQueueReceive()s . . . 153
xQueueReceiveFromISR() . . . 159
xQueueSelectFromSet() . . . 153
xQueueSend() . . . 159, 163
xQueueSendFromISR() . . . 146, 159
xSemaphoreCreateMutex() . . . 165, 167

xSemaphoreGive() 165, 167
xSemaphoreGiveFromISR() 132, 144
xSemaphoreTake() 144, 149, 165, 167
xTaskCreate() 130, 137, 163
xTaskNotify() 154
xTaskNotifyFromISR() 154
xTaskNotifyGive() 154
xTaskNotifyWait() 154