

---

# Langage de programmation Systèmes embarqués

---

Application au STM32F401

Licence EPOCS (UE23)

Pierre Pardo  
[pierre.pardo@univ-amu.fr](mailto:pierre.pardo@univ-amu.fr)

2020-2021



---

**Notes**



# Table des matières

<b>Table des matières</b>	<b>3</b>
<b>1 Cortex-M, STM32F401RE et carte d'extension</b>	<b>7</b>
1.1 Familles de microcontrôleurs . . . . .	7
1.2 Configuration matérielle utilisée . . . . .	9
1.3 Le STM32F401 . . . . .	12
1.4 La carte NUCLEO-F401RE . . . . .	18
1.5 La carte MW000505A . . . . .	19
1.6 Architecture du STM32F401 . . . . .	20
<b>2 Du hardware au software</b>	<b>45</b>
2.1 De l'analogique au numérique . . . . .	45
2.2 L'architecture du point de vue du développeur . . . . .	47
2.3 Les instructions assembleurs . . . . .	53
2.4 Traitement des instructions . . . . .	54
<b>3 Développement</b>	<b>61</b>
3.1 Notions d'IDE . . . . .	61
3.2 Utilisation de STM32CubeIDE . . . . .	63
3.3 Bibliothèques CMSIS, StdPeriph, HAL, LL, . . . . .	64
3.4 Utilisation du programmeur autonome de STM32 : STM32 ST-LINK Utility . . . . .	68
3.5 Cycle développement / compilation / programmation . . . . .	69
3.6 Méthodes agiles . . . . .	70
<b>4 Notions de temps réel</b>	<b>75</b>
4.1 Pourquoi un système d'exploitation temps réel? . . . . .	75
4.2 Quelques définitions . . . . .	76
<b>5 Temps réel Multitâches avec FreeRTOS</b>	<b>83</b>
5.1 Concepts de base . . . . .	83

## Table des matières

---

5.2	Systick et <b>FreeRTOS</b> . . . . .	90
5.3	Tâche et descripteur de tâche . . . . .	91
5.4	Changement de contexte . . . . .	96
5.5	Ordonnancement des tâches et préemption . . . . .	97
5.6	Cycle de vie des tâches . . . . .	99
<b>6</b>	<b>Gestion de la mémoire</b> . . . . .	<b>107</b>
6.1	Problème d'allocation et contraintes de l'embarqué . . . . .	107
6.2	Gestionnaires mémoire . . . . .	108
6.3	Erreurs liées à la mémoire . . . . .	109
<b>7</b>	<b>Ressources critiques, MutEx, synchro, ...</b> . . . . .	<b>111</b>
7.1	Gestion des ressources . . . . .	111
7.2	Primitives de synchronisation . . . . .	122
7.3	Communication entre tâches et Queues de communication . . . . .	126
7.4	De nouveaux concepts (v10) . . . . .	136
<b>8</b>	<b>Interruptions, timers</b> . . . . .	<b>139</b>
8.1	Gestion des interruptions . . . . .	139
8.2	Timer . . . . .	147
<b>9</b>	<b>Trace, débogage et analyse</b> . . . . .	<b>149</b>
9.1	Instrumentation manuelle . . . . .	150
9.2	Instrumentation intégrée dans <b>FreeRTOS</b> . . . . .	151
9.3	Utilisation de <b>STM-STUDIO-STM32</b> . . . . .	155
<b>A</b>	<b>Annexe</b> . . . . .	<b>157</b>
A.1	Configuration de <b>FreeRTOS</b> . . . . .	157
A.2	Définitions . . . . .	158
A.3	Types de données et limites . . . . .	159
A.4	Erreurs courantes . . . . .	159
A.5	Mots-clefs du langage C . . . . .	162
A.6	Aide-mémoire sur les registres <b>GPIO</b> . . . . .	162
A.7	Aide-mémoire <b>API FreeRTOS</b> . . . . .	164
A.8	Table des caractères ASCII . . . . .	167
<b>B</b>	<b>Index</b> . . . . .	<b>169</b>
B.1	Table des figures . . . . .	169
B.2	Liste des tableaux . . . . .	170
B.3	Liste des programmes . . . . .	170
	<b>Bibliographie</b> . . . . .	<b>173</b>

---

## Introduction

Ce document rassemble les informations nécessaires pour aborder la programmation d'un microcontrôleur **STM32F401** en utilisant un système d'exploitation temps réel **FreeRTOS**.

Ce n'est pas un manuel de référence exhaustif, pour cela on pourra se référer pour la partie matérielle au manuel de référence du **STM32F401** [8], et à son manuel de programmation [10].

Pour la partie logicielle, les documents utiles seront le manuel de référence de **FreeRTOS** [6] et l'ouvrage du créateur de **FreeRTOS** [3].

Nous allons aborder l'architecture du **STM32F401** et analyser le hardware que nous allons utiliser pour nos tests, puis nous examinerons les concepts clefs des système d'exploitation temps réels. Nous mettrons ensuite en œuvre **FreeRTOS** pour résoudre les problèmes classiques

Le langage **C** utilisé pour la programmation est supposé connu, toutefois, quelques rappels seront distillés dans ce document quand des points critiques ou mal défini du langage seront abordés. On pourra toujours se référer à [4].



### Commentaires

Tous les commentaires sont les bien venus, et j'essaierais de répondre à toutes les questions.

✉ [pierre.pardo@univ-amu.com](mailto:pierre.pardo@univ-amu.com) ou ✉ [ppardo@metraware.com](mailto:ppardo@metraware.com)





# 1

## Cortex-M, STM32F401RE et carte d'extension

*Ce chapitre traite de l'architecture des microcontrôleurs, et présente le matériel qui va être utilisé.*

*“When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.”*

- Edsger Dijkstra,

### 1.1 Familles de microcontrôleurs

La plupart des fondeurs de microcontrôleurs disposent de leur propres architectures, mais certains utilisent une architecture licenciée à la société ARM<sup>1</sup>. On la retrouve dans une grande partie des smartphones et tablettes (85% du marché), ainsi que sur de grandes gammes de microcontrôleurs (environ 15 milliards d'équipement à base d'un cœur d'ARM vendus en 2015).

Chez chaque fondeur (ayant sa propre architecture, ou utilisant une licence ARM), nous allons retrouver des gammes de microcontrôleurs dédiés à différentes applications :

- Des microcontrôleurs faible consommation,
- Des microcontrôleurs possédant un DSP,
- Des microcontrôleurs possédant une unité à virgule flottante,
- Des microcontrôleurs possédant des bus particuliers (en plus ou moins grande quantité)
  - CAN,
  - SPI,

---

1. Acor RISC Machine puis Advanced RISC Machine

- I<sup>2</sup>C,
- UARTs
- Cryptographie,
- Ethernet,
- ...
- Des packages différents pour s'adapter aux nombres d'entrées/sorties nécessaires,
- Des gammes de températures,
- ...

A titre d'exemple, chez Microchip, il y a 422 références dans la gamme des PIC 8 bits, et presque 200 dans la gamme des PIC 32 bits.

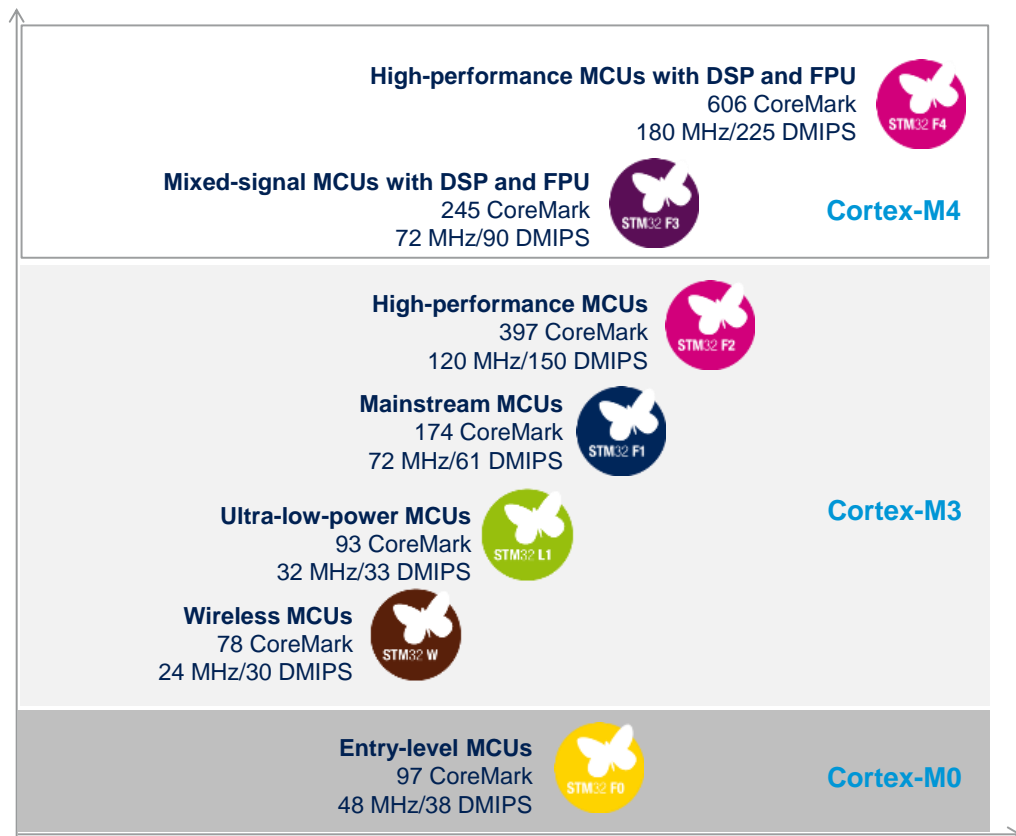


FIGURE 1.1 – Offre des STM32

Chez ST, pour la gamme des STM32 (basée sur des cœurs ARM M0/M0+, M3, M4 et M7) il y a 684 références.

La figure 1.1 montre la répartition de la gamme des STM32.

Les architectures **ARM** se déclinent en gamme **A** (tablette et smartphone), **M** (microcontrôleurs pour l'embarqué), **R** (extension temps réel).

Nous allons utiliser un **ARMv7-M**. La liste suivante indique les relations entre architecture et cœur :

- **ARMv6-M**
  - Cortex M0, M0+
  - Cortex M1
- **ARMv7-M**
  - Cortex M3
  - Cortex M4, M4F
  - Cortex M7
- **ARMv8-M**
  - Cortex M23
  - Cortex M33, M33F
  - Cortex M35P, M35PF

## 1.2 Configuration matérielle utilisée

Nous allons utiliser pour nos manipulations un microcontrôleur ayant un cœur **ARM Cortex-M4F** 32 bit. Ce composant se trouve sur une carte de test **NUCLEO-F401RE**, qui simplifie le développement et le débogage d'application. Pour étendre les entrées sorties de la carte **NUCLEO** (qui ne dispose que d'un bouton poussoir et d'une led), nous allons connecter une carte d'extension (voir figure 1.3) grâce aux ports compatibles **Arduino**.

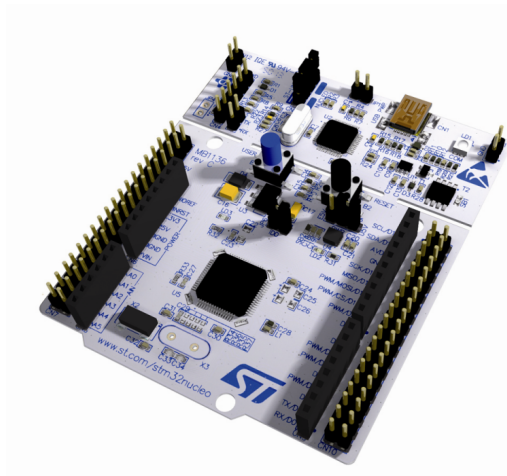


FIGURE 1.2 – Carte NUCLEO-F401RE

La carte d'extension va nous permettre de tester plus de possibilités d'entrées sorties que la carte **NUCLEO-F401RE**. La carte d'extension est décrite au chapitre 1.5, une photo de la carte d'extension avec la carte **NUCLEO** se trouve en figure 1.10.

## 1.2 Configuration matérielle utilisée

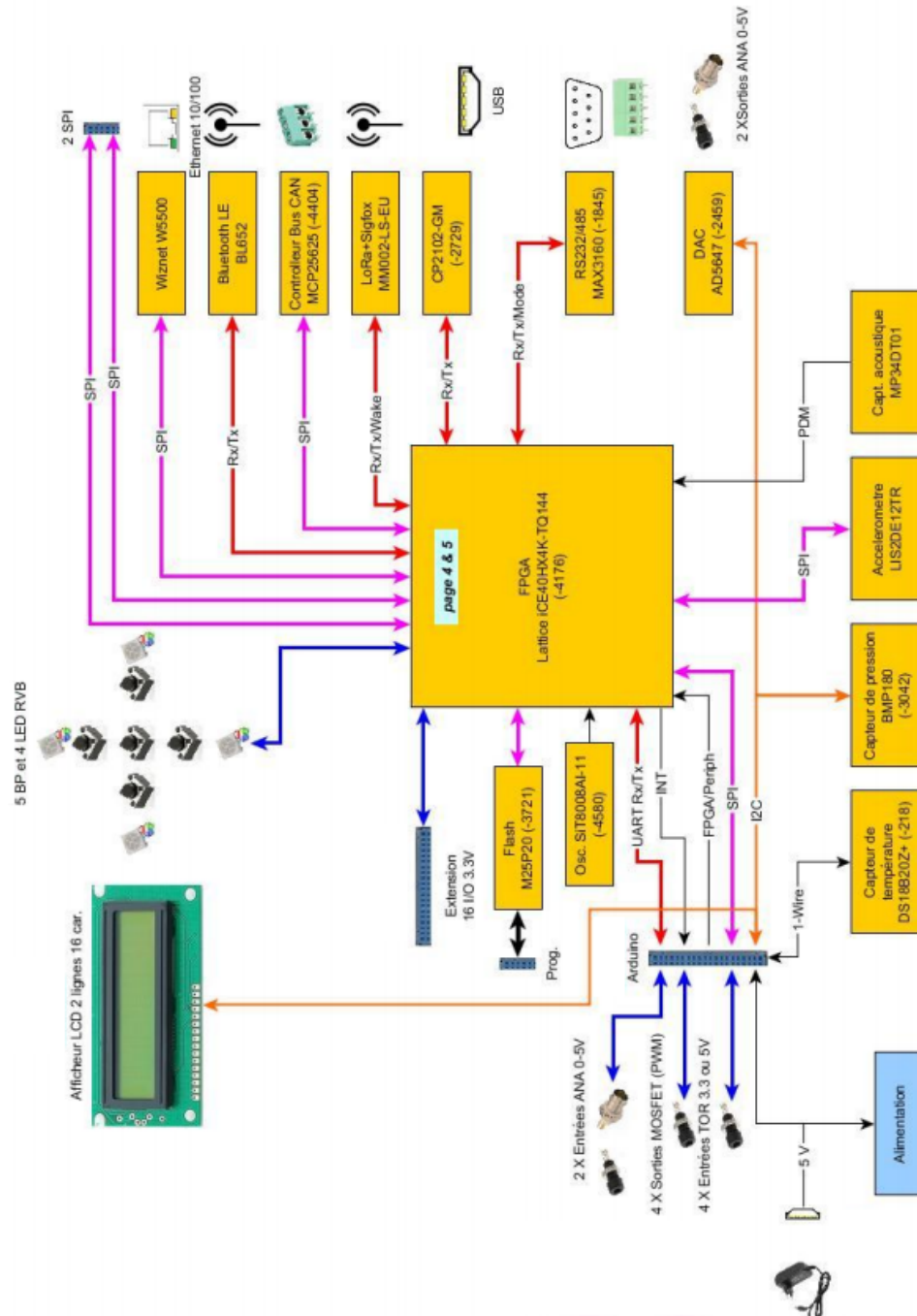


FIGURE 1.3 – Synoptique de la carte d'extension

## 1.3 Le STM32F401

Des informations complètes se trouvent dans les documents [10] et [8]. Le document [2] contient la documentation générique des Cortex-M4.

### 1.3.1 Le STM32F401RE en quelques mots

- Performance : Le STM32F401RE fonctionne à 84 MHz, il possède un DSP et une unité de traitement à virgule flottante.
- Consommation : La consommation est de 146  $\mu$ A/MHz en fonctionnement, et de 9  $\mu$ A en mode stoppé.
- Intégration et périphériques :
  - 512 kB de Flash
  - 96 kB de RAM
  - 3x USARTs configurable max 10.5 Mbit/s
  - 4x SPI configurable max 42 Mbit/s
  - 3x I<sup>2</sup>C,
  - 1x SDIO (mmc, sd card, ...),
  - 1x USB 2.0 OTG full speed,
  - 2x full duplex I<sup>2</sup>S (max 32 bit à 192 kHz),
  - 12-bit ADC à 2.4 MSPS,
  - 10 timers, 16- et 32-bit, (max 84 MHz)
  - ...
- Prix : 6 € l'unité

### 1.3.2 Différents boîtiers

Le STM32F401 est disponible en plusieurs tailles et technologies de boîtier.

**UQFN48** 7 × 7 mm

**WLCS49** 3.06 × 3.06 mm

**LQFP64** 10 × 10 mm

**LQFP100** 14 × 14 mm

**UFBGA100** 7 × 7 mm

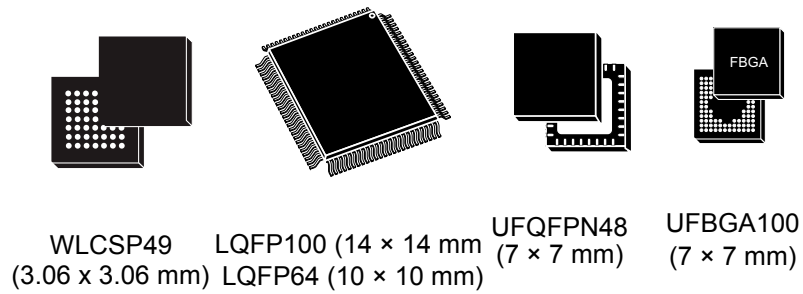


FIGURE 1.4 – Boîtiers STM32F401RE

Il y a un multiplexage des fonctions avec les broches du composant. Toutes les fonctions ne sont pas toujours disponibles. Il faut consulter le datasheet [\[9\]](#) pour vérifier les disponibilités des fonctions.

La figure 1.5 montre un extrait de la table d'affectation des fonctions suivant le boîtier.

Pin Number					Pin name (function after reset) <sup>(1)</sup>	Pin type	I/O structure	Notes	Alternate functions	Additional functions
UQFN48	WLCSP49	LQFP64	LQFP100	UFBGA100						
-	-	39	65	E10	PC8	I/O	FT	-	USART6_CK, TIM3_CH3, SDIO_D0, EVENTOUT	-
-	-	40	66	D12	PC9	I/O	FT	-	I2S_CKIN, I2C3_SDA, TIM3_CH4, SDIO_D1, MCO_2, EVENTOUT	-
29	D1	41	67	D11	PA8	I/O	FT	-	I2C3_SCL, USART1_CK, TIM1_CH1, OTG_FS_SOF, MCO_1, EVENTOUT	-
30	D2	42	68	D10	PA9	I/O	FT	-	I2C3_SMBA, USART1_TX, TIM1_CH2, EVENTOUT	OTG_FS_VBUS
31	C2	43	69	C12	PA10	I/O	FT	-	USART1_RX, TIM1_CH3, OTG_FS_ID, EVENTOUT	-
32	C1	44	70	B12	PA11	I/O	FT	-	USART1_CTS, USART6_TX, TIM1_CH4, OTG_FS_DM, EVENTOUT	-
33	C3	45	71	A12	PA12	I/O	FT	-	USART1_RTS, USART6_RX, TIM1_ETR, OTG_FS_DP, EVENTOUT	-
34	B3	46	72	A11	PA13 (JTMS-SWDIO)	I/O	FT	-	JTMS-SWDIO, EVENTOUT	-
-	-	-	73	C11	VCAP2	S	-	-	-	-
35	B1	47	74	F11	VSS	S	-	-	-	-
36	-	48	75	G11	VDD	S	-	-	-	-
-	B2	-	-	-	VDD	S	-	-	-	-

FIGURE 1.5 – Fonctions disponibles suivants les boîtiers (extrait)

### 1.3.3 Nommage

Nous allons utiliser un STM32F401RET6.

**STM32** microcontrôleur ST ARM 32 bit

**F** standard

**401** famille

**pin count** C=48/49 R=64 V=100

**taille mémoire flash** D=384 K octets E=512 K octets D=384 kB E=512 kB

**package** H=UFBGA T=LQFP U=UFQFPN Y=WLCSP

**gamme température** 6=industrielle (−40 °C..85 °C)



### 1.3.4 STM32F401RE Pinout (boîtier LQFP64)

Le boîtier présent sur la carte NUCLE0-F401RE est un LQFP64. Pour un nouveau design à base de STM32F401, il faudra vérifier que toutes les fonctions nécessaires à l'application sont bien présentes sur le boîtier choisi.

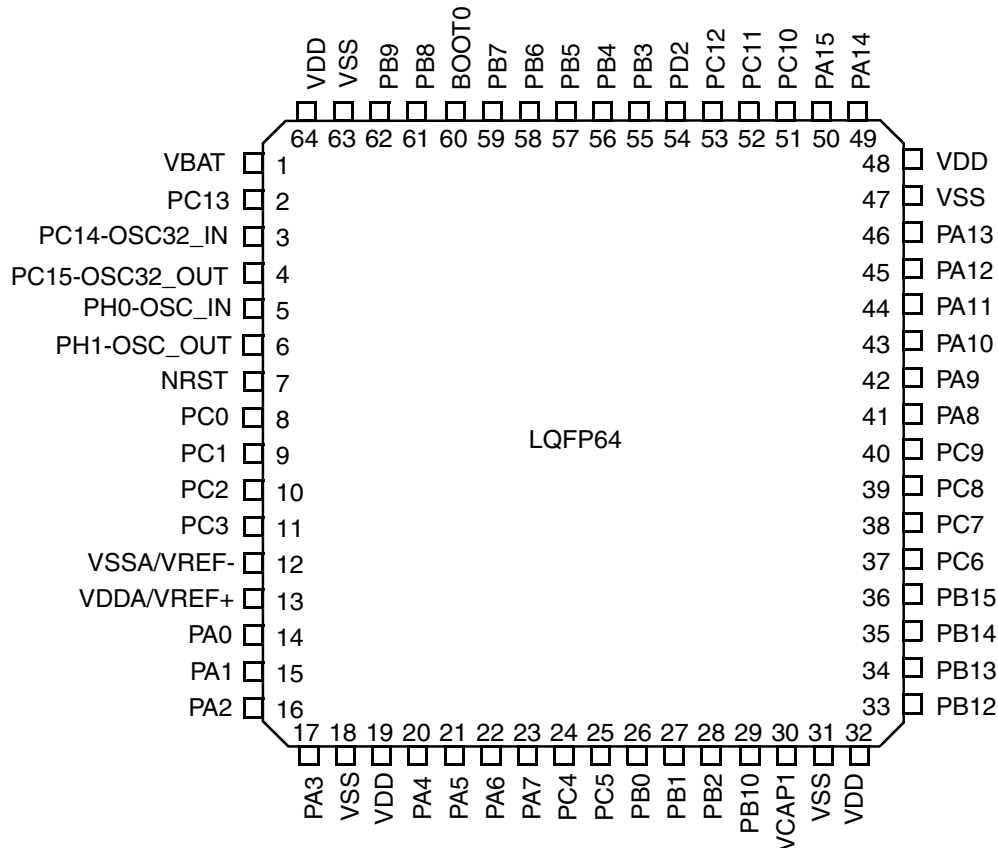


FIGURE 1.6 – Pinout STM32F401RET

### 1.3.5 STM32F401xD/xE diagramme des fonctionnalités

Le diagramme (figure 1.7) expose les fonctions disponibles, regroupées par catégories pour la gamme des STM32F4. Le STM32F401 est un microcontrôleur généraliste qui est utilisé dans des domaines variés d'applications.



Common core peripherals and architecture:

Communication peripherals: USART, SPI, I <sup>2</sup> C
Multiple general-purpose timers
Integrated reset and brown-out warning
Multiple DMA
2x watchdogs
Real-time clock
Integrated regulator
PLL and clock circuit
External memory interface (FSMC)
Up to 3x 12-bit DAC
Up to 4x 12-bit ADC (Up to 5 MSPS)
Main oscillator and 32 kHz oscillator
Low-speed and high-speed internal RC oscillators
-40 to +85 °C and up to 105 °C operating temperature range
Low voltage 2.0 to 3.6 V or 1.65/1.7 to 3.6 V (depending on series)
Temperature sensor

STM32 F4 series - High performance with DSP (STM32F401/405/415/407/417/427/437/429/439)										
180 MHz Cortex-M4 with DSP and FPU	Up to 256-Kbyte SRAM	Up to 2-Mbyte Flash	2x USB 2.0 OTG FS/HS	3-phase MC timer	2x CAN 2.0B	SDIO 2x I <sup>2</sup> S audio Camera IF	Ethernet IEEE 1588	Crypto TFT LCD + SDRAM		
STM32 F3 series - Mixed-signal with DSP (STM32F302/303/313/372/373/383)										
72 MHz Cortex-M4 with DSP and FPU	Up to 48-Kbyte SRAM & CCM-SRAM	Up to 256-Kbyte Flash	USB 2.0 FS	2x 3-phase MC timer (144 MHz)	CAN 2.0B	Up to 7x comparator	3x 16-bit ΣΔ ADC	4x PGA		
STM32 F2 series - High performance (STM32F205/215/207/217)										
120 MHz Cortex-M3 CPU	Up to 128-Kbyte SRAM	Up to 1-Mbyte Flash	2x USB 2.0 OTG FS/HS	3-phase MC timer	2x CAN 2.0B	SDIO 2x I <sup>2</sup> S audio Camera IF	Ethernet IEEE 1588	Crypto		
STM32 F1 series - Mainstream - 5 product lines (STM32F100/101/102/103 and 105/107)										
Up to 72 MHz Cortex-M3 CPU	Up to 96-Kbyte SRAM	Up to 1-Mbyte Flash	USB 2.0 OTG FS	3-phase MC timer	Up to 2x CAN 2.0B	SDIO 2x I <sup>2</sup> S audio	Ethernet IEEE 1588			
STM32 F0 series - Entry level (STM32F030/50/051)										
48 MHz Cortex-M0 CPU	Up to 8-Kbyte SRAM	Up to 64-Kbyte Flash	3-phase MC timer	Comparator	CEC					
STM32 L1 series - Ultra-low-power (STM32L100/151/152/162)										
32 MHz Cortex-M3 CPU	Up to 48-Kbyte SRAM	Up to 384-Kbyte Flash	USB FS device	Up to 12-Kbyte EEPROM	LCD 8x40 4x44	Comparator	BOR MSI VScal	AES 128-bit		
STM32 W series - Wireless (STM32W108)										
24 MHz Cortex-M3 CPU	Up to 16-Kbyte SRAM	Up to 256-Kbyte Flash	2.4 GHz IEEE 802.15.4 Transceiver	Lower MAC Digital baseband	AES 128-bit					

FIGURE 1.8 – Gamme des STM32



- 1 lien usb (identifié comme stockage de masse, liaison série et port de débogage par le PC)

L'utilisation de connecteurs d'extension standard permet très rapidement d'augmenter les fonctionnalités de la carte **NUCLEO**. C'est ce que nous avons fait pour créer la carte d'extension qui sera présentée au chapitre 1.5.

On trouve pour quelques euros des cartes d'extension (appelées **Shield**) pour ajouter des fonctionnalités :

- réseau ethernet, bluetooth, LoRa, ...
- des entrées/sorties supplémentaires
- lecteur de cartes SD
- contrôleur d'axes
- ...

## 1.5 La carte MW000505A

La figure 1.10 montre l'empilement de la carte d'extension sur la carte **NUCLEO**. Le synoptique se trouve sur la figure 1.3.

La carte d'extension apporte les connexions suivantes sur le port **Arduino** :

- Une liaison série connectée au **FPGA**
- Une liaison **SPI** connectée au **FPGA**, cela permet de piloter les périphériques du **FPGA** indirectement :
  - Des boutons poussoirs (5) et des leds **RVB** (4).
  - De la flash **M25P20**
  - Des liaisons **SPI** externes
  - Un module de communication **Wiznet** (ethernet)
  - Un module **Bluetooth Low Energy**
  - Un contrôleur de bus **CAN**
  - Un module **LoRa + Sigfox**
  - Un contrôleur **USB**
  - Une liaison **RS232/485**
  - Un capteur acoustique **MP34DT01**
  - Un accéléromètre **LIS2DE12TR**
  - Des **GPIO** (16 \* 3.3 V)
- Une liaison **I2C** connectée à :
  - Un capteur de pression **BMP180**
  - Un convertisseur **DAC AD5647**
  - Un afficheur **LCD**
- Une liaison 1-wire connectée à un capteur de température **DS18B20Z**
- Des entrées analogiques (x2)
- Des sorties **PWM** (x4)

- Des entrées TOR (x4)

Les périphériques de la carte d'extension connectés au FPGA se pilotent par l'intermédiaire de la liaison SPI. Les registres nécessaires sont décrits dans [5] et des exemples se trouvent dans les TD.

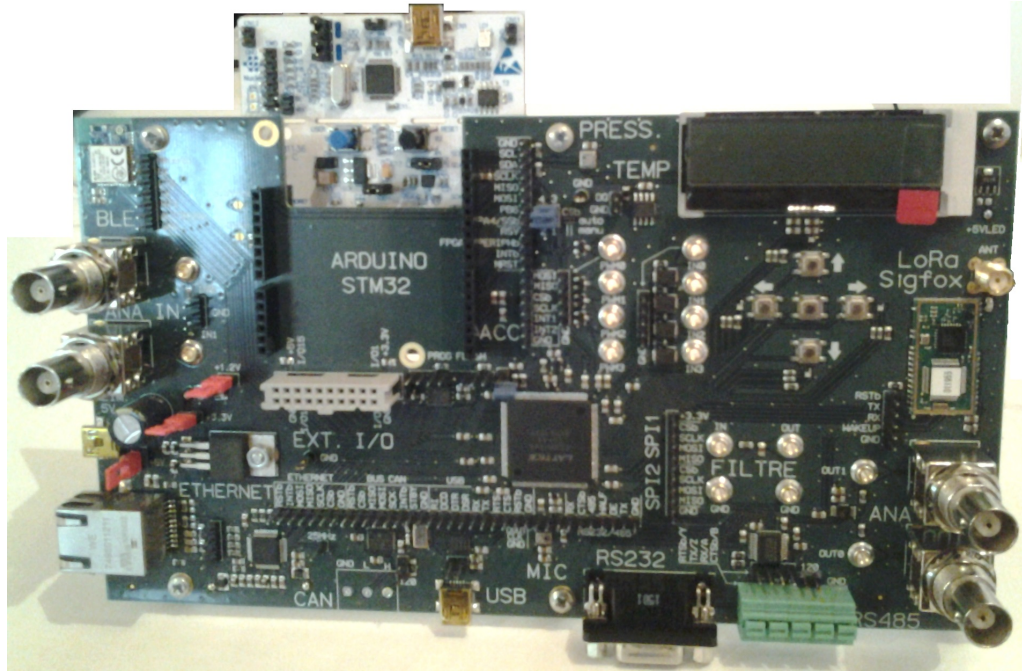


FIGURE 1.10 – Carte d'extension et carte NUCLEO

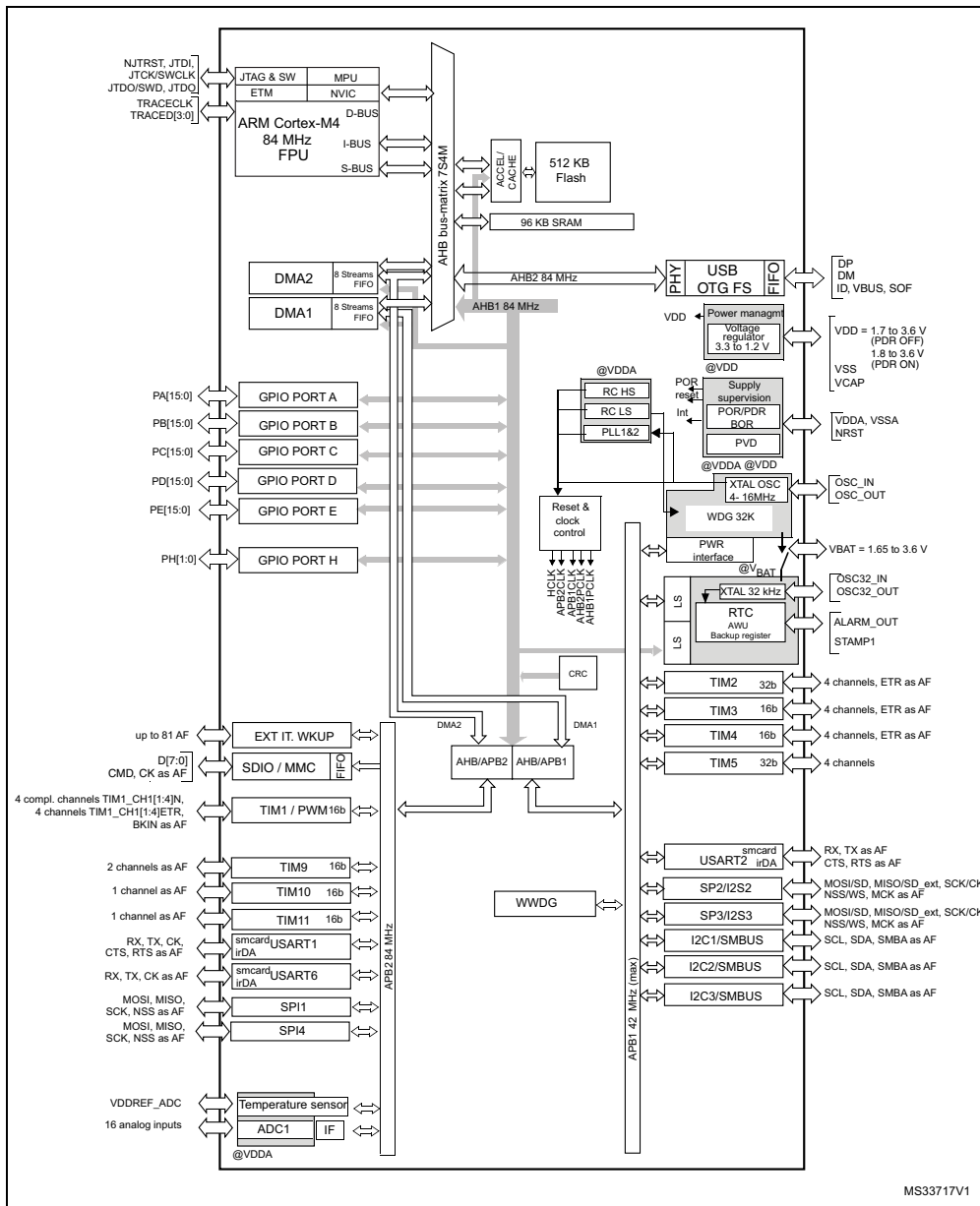
## 1.6 Architecture du STM32F401

### 1.6.1 STM32F401xD/xE block diagram

La figure 1.11 montre le diagramme bloc du STM32F401. Les deux contrôleurs de DMA permettent de soulager le cœur en laissant les périphériques avoir un accès direct avec la mémoire.

Un exemple d'utilisation du DMA se trouve dans le listing 1.7 et dans les TD.





1. The timers connected to APB2 are clocked from TIMxCLK up to 84 MHz, while the timers connected to APB1 are clocked from TIMxCLK up to 42 MHz.

FIGURE 1.11 – Diagramme bloc du STM32F401

### 1.6.2 Mémoire et bus

Les bus sont organisés suivant la matrice figure 1.12. Cela permet des accès simultanés, à des fréquences différentes.

Les bus ont une taille de 32 bits.

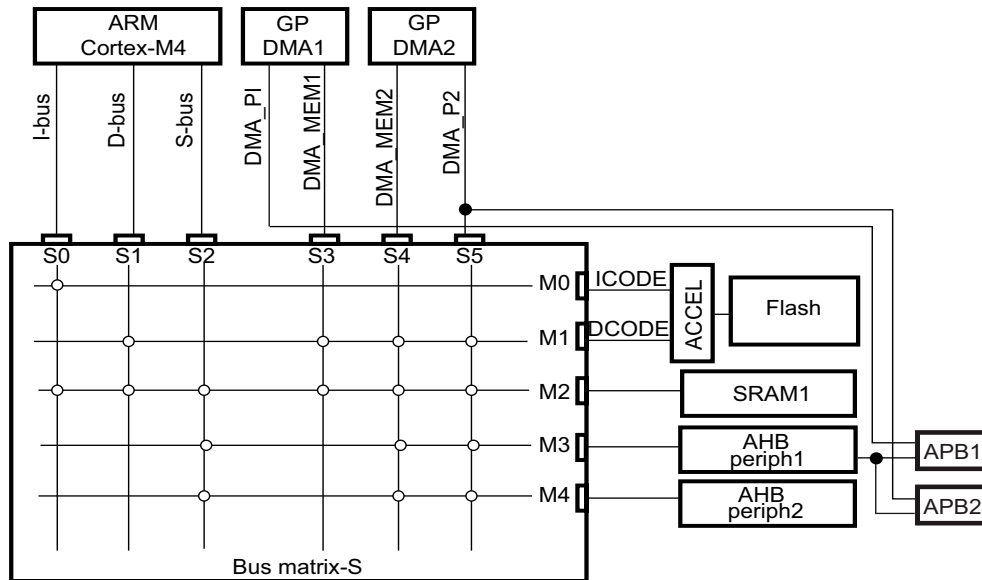


FIGURE 1.12 – Mémoire et bus

### 1.6.2.1 Bus instructions / data système

**I-Bus** Instruction bus. Utilisé par le cœur pour récupérer les instructions.

Cibles de ce bus :

- Flash interne
- SRAM

**D-Bus** Data bus. Chargement de valeurs et accès debug. Cibles de ce bus :

- Flash interne
- SRAM

**S-Bus** System bus. Accès aux données situées dans un périphérique ou en SRAM. Cibles de ce bus :

- SRAM
- Les périphériques sur les bus AHB1, APB, AHB2

C'est une architecture Harvard, avec un bus pour les instructions et un bus pour les données.

### 1.6.2.2 Bus des périphériques

**AHB** Advanced High Performance Bus

**APB** Advanced Peripheral Bus



### 1.6.2.3 Bus DMA

**DMA memory bus** Utilisé par le DMA pour des transferts vers/depuis la mémoire. Cibles de ce bus :

- Flash interne
- SRAM
- (sur S4) Les périphériques AHB1, APB1, AHB2, APB2

**DMA peripheral bus** Utilisé pour des transferts vers les périphériques ou pour des transferts mémoire  $\rightleftharpoons$  mémoire. Cibles de ce bus :

- Les périphériques connectés aux bus AHB1, APB1, AHB2, APB2
- Flash interne
- SRAM

## 1.6.3 Organisation mémoire

Toute la mémoire (programme, data, registres, périphérique, ...) est organisée linéairement sur un espace d'adressage de 32 bits. La mémoire totale adressable est donc de 4 GB, la figure 1.13 montre l'organisation de la mémoire.

Il ne faut pas accéder aux adresses notées **Reserved**.

Les données sont stockées en **Little Endian**.

L'octet de poids faible est à la première adresse, le reste suit...

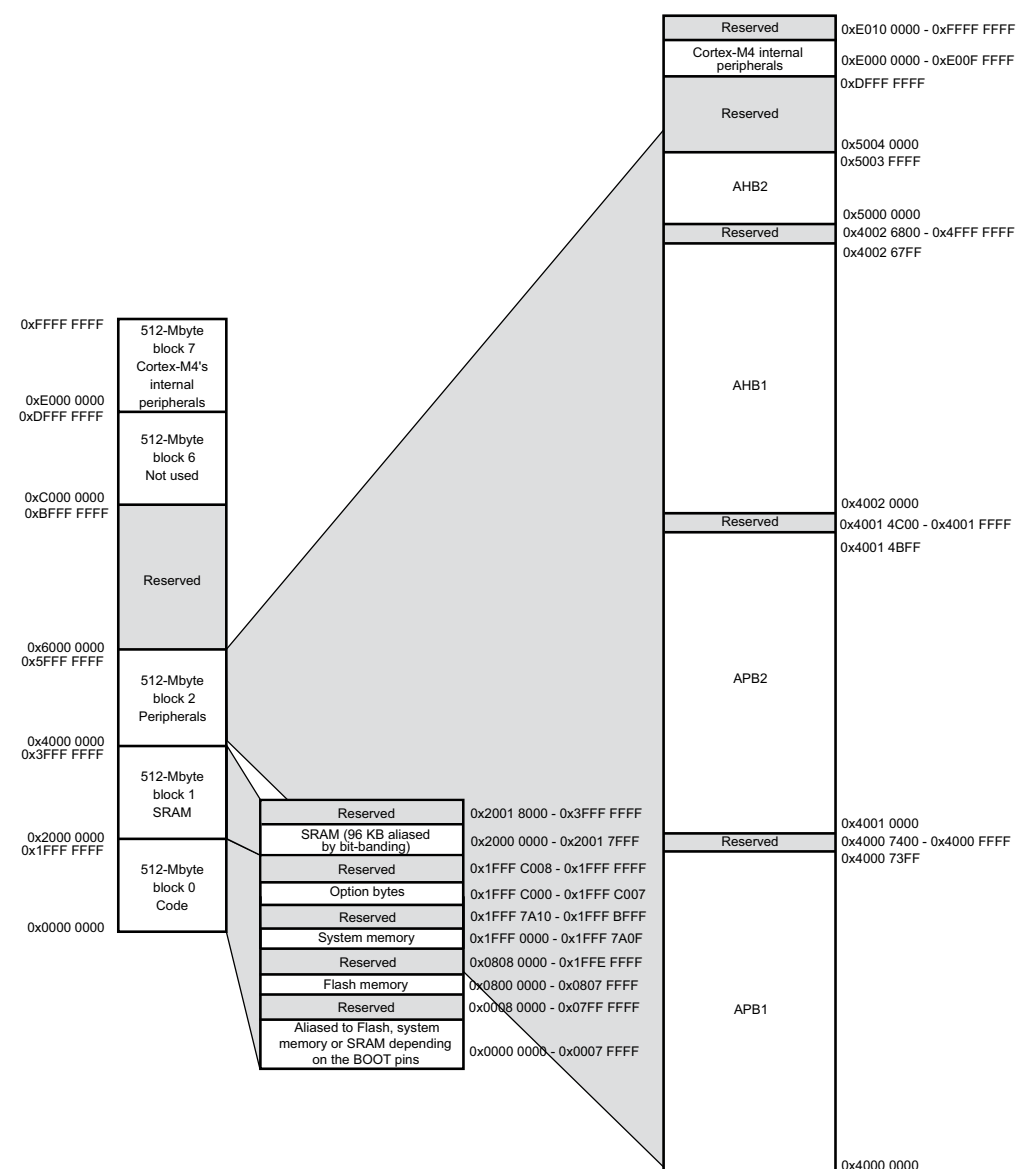


FIGURE 1.13 – Organisation mémoire

Les figures 1.14, 1.15 et 1.16 montrent la répartition des adresses pour les périphériques.

## 1.6 Architecture du STM32F401

Bus	Boundary address	Peripheral
	0xE010 0000 - 0xFFFF FFFF	Reserved
Cortex <sup>®</sup> -M4	0xE000 0000 - 0xE00F FFFF	Cortex-M4 internal peripherals
	0x5004 0000 - 0xDFFF FFFF	Reserved
AHB2	0x5000 0000 - 0x5003 FFFF	USB OTG FS
AHB1	0x4002 6800 - 0x4FFF FFFF	Reserved
	0x4002 6400 - 0x4002 67FF	DMA2
	0x4002 6000 - 0x4002 63FF	DMA1
	0x4002 5000 - 0x4002 4FFF	Reserved
	0x4002 3C00 - 0x4002 3FFF	Flash interface register
	0x4002 3800 - 0x4002 3BFF	RCC
	0x4002 3400 - 0x4002 37FF	Reserved
	0x4002 3000 - 0x4002 33FF	CRC
	0x4002 2000 - 0x4002 2FFF	Reserved
	0x4002 1C00 - 0x4002 1FFF	GPIOH
	0x4002 1400 - 0x4002 1BFF	Reserved
	0x4002 1000 - 0x4002 13FF	GPIOE
	0x4002 0C00 - 0x4002 0FFF	GIPOD
	0x4002 0800 - 0x4002 0BFF	GPIOC
	0x4002 0400 - 0x4002 07FF	GPIOB
	0x4002 0000 - 0x4002 03FF	GPIOA

FIGURE 1.14 – Périphériques et adresses mémoire

Bus	Boundary address	Peripheral
APB2	0x4001 4C00- 0x4001 FFFF	Reserved
	0x4001 4800 - 0x4001 4BFF	TIM11
	0x4001 4400 - 0x4001 47FF	TIM10
	0x4001 4000 - 0x4001 43FF	TIM9
	0x4001 3C00 - 0x4001 3FFF	EXTI
	0x4001 3800 - 0x4001 3BFF	SYSCFG
	0x4001 3400 - 0x4001 37FF	SPI4/I2S4
	0x4001 3000 - 0x4001 33FF	SPI1
	0x4001 2C00 - 0x4001 2FFF	SDIO
	0x4001 2400 - 0x4001 2BFF	Reserved
	0x4001 2000 - 0x4001 23FF	ADC1
	0x4001 1800 - 0x4001 1FFF	Reserved
	0x4001 1400 - 0x4001 17FF	USART6
	0x4001 1000 - 0x4001 13FF	USART1
	0x4001 0800 - 0x4001 0FFF	Reserved
	0x4001 0400 - 0x4001 07FF	TIM8
	0x4001 0000 - 0x4001 03FF	TIM1
	0x4000 7400 - 0x4000 FFFF	Reserved

FIGURE 1.15 – Périphériques et adresses mémoire (suite)

Bus	Boundary address	Peripheral
APB1	0x4000 7000 - 0x4000 73FF	PWR
	0x4000 6000 - 0x4000 6FFF	Reserved
	0x4000 5C00 - 0x4000 5FFF	I2C3
	0x4000 5800 - 0x4000 5BFF	I2C2
	0x4000 5400 - 0x4000 57FF	I2C1
	0x4000 4800 - 0x4000 53FF	Reserved
	0x4000 4400 - 0x4000 47FF	USART2
	0x4000 4000 - 0x4000 43FF	I2S3ext
	0x4000 3C00 - 0x4000 3FFF	SPI3 / I2S3
	0x4000 3800 - 0x4000 3BFF	SPI2 / I2S2
	0x4000 3400 - 0x4000 37FF	I2S2ext
	0x4000 3000 - 0x4000 33FF	IWDG
	0x4000 2C00 - 0x4000 2FFF	WWDG
	0x4000 2800 - 0x4000 2BFF	RTC & BKP Registers
	0x4000 1000 - 0x4000 27FF	Reserved
	0x4000 0C00 - 0x4000 0FFF	TIM5
	0x4000 0800 - 0x4000 0BFF	TIM4
	0x4000 0400 - 0x4000 07FF	TIM3
	0x4000 0000 - 0x4000 03FF	TIM2

FIGURE 1.16 – Périphériques et adresses mémoire (suite)

### 1.6.3.1 Mécanisme de Bit-Banding

La mémoire est composée de mots de 32 bits.

L'accès (en lecture ou écriture) à un seul bit parmi 32 nécessite plusieurs opérations.

Pour une écriture par exemple :

1. Lecture du mot de 32 bits
2. Modification du bit (par des masquages & et |)
3. Écriture du mot de 32 bits en mémoire

Cette procédure n'est pas sûre si des interruptions peuvent survenir. Il faudra protéger en contrôlant les interruptions.

Avec un système d'exploitation temps réel, nous parlerons de sections critiques. Nous verrons les mécanismes de protection en 7.1.

Le principe du Bit-Banding est de permettre un accès atomique à un seul bit parmi les 32. Pour cela une zone mémoire 32 fois plus importante que la zone d'origine est accessible (comme indiqué sur la figure 1.17). En accédant à une de ces adresses, on accède directement à un des bits de la zone d'origine.

Il n'y a que 2 zones de bit-banding, une pour les périphériques, et une pour la SRAM.

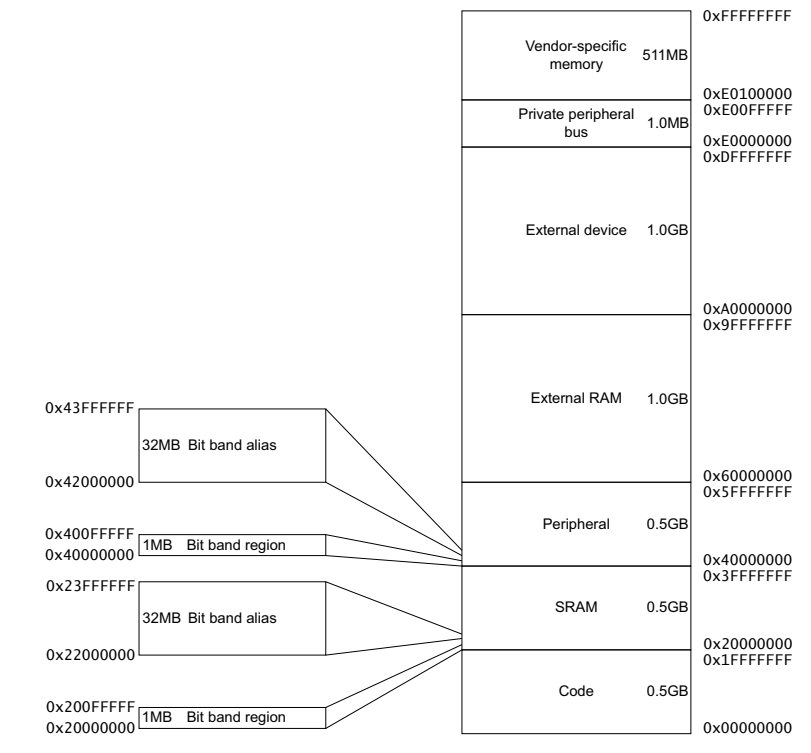


FIGURE 1.17 – Plan mémoire avec bit-banding

- Uniquement en SRAM
  - $0x20000000 - 0x200FFFFF \Rightarrow 0x22000000 - 0x23FFFFFF$
- Et pour les périphériques
  - $0x40000000 - 0x400FFFFF \Rightarrow 0x42000000 - 0x43FFFFFF$

Le bit-banding ne fonctionne pas en DMA.

Calcul de l'adresse :

$\text{bit\_word\_addr} = \text{bit\_band\_base} + (\text{byte\_offset} * 32) + (\text{bit\_number} * 4)$

Pour accéder au bit 5 du registre ODR de GPIOA par exemple :

- $\text{bit\_band\_base} = 0x42000000$  (pour les périphériques)
- $\text{byte\_offset} = \text{GPIOA} \rightarrow \text{ODR} ((0x20000 + 0x14) * 32 = 0x400280)$
- $\text{bit\_number} = 5$  ( $5 * 4 = 20 = 0x14$ )

Il est équivalent de :

- Faire des masquages en  $\text{GPIOA} \rightarrow \text{ODR}$  avec
  - $\& \sim(1 \ll 5)$
  - $| (1 \ll 5)$
- Accéder à l'adresse  $0x42400294$

Mais le bit-banding est :

- Plus sûr (atomique)

- Plus rapide

Le listing 1.1 montre l'utilisation du Bit-banding pour accéder à un bit. Ici, cela permet de définir l'état du bit 5 du GPIOA.



### Bit-banding sur les sorties GPIO

Pour les GPIO, les registres BSRRL et BSRRH permettent d'effectuer des modifications d'un ou plusieurs bit(s) directement, sans risque des problèmes inhérents au registre ODR.

```

1  /* bitband type */
2  typedef volatile uint32_t * const bitband_t;
3
4  /* base address for bit banding */
5  #define BITBAND_SRAM_REF          (0x20000000)
6  /* base address for bit banding */
7  #define BITBAND_SRAM_BASE        (0x22000000)
8  /* base address for bit banding */
9  #define BITBAND_PERIPH_REF       (0x40000000)
10 /* base address for bit banding */
11 #define BITBAND_PERIPH_BASE      (0x42000000)
12
13 /* sram bit band */
14 #define BITBAND_SRAM(address, bit) ((void*)(BITBAND_SRAM_BASE + \
15      (((uint32_t)address) - BITBAND_SRAM_REF) * 32 + (bit) * 4))
16
17 /* periph bit band */
18 #define BITBAND_PERIPH(address, bit) ((void*)(BITBAND_PERIPH_BASE + \
19      (((uint32_t)address) - BITBAND_PERIPH_REF) * 32 + (bit) * 4))
20
21
22 /* definition of bit band GPIOA.ODR, bit 5*/
23 bitband_t bo = BITBAND_PERIPH(&GPIOA->ODR, 5);
24
25
26 /* bit set to 0, atomic */
27 (*bo) = 0;
28
29 /* bit set to 1, atomic */
30 (*bo) = 1;

```

Listing 1.1 – Macro pour le Bit-banding

## 1.6.4 Contrôleur de reset et d'horloge RCC

La définition de tous les registres relatifs au RCC se trouve en [8] (chapitre 6.3).

#### 1.6.4.1 Contrôleur de reset

Un reset système peut se produire sur une des causes suivantes :

- Reset externe (pin NRST)
- Un watchdog (WWDG ou IWDG)
- Reset interne (software, en définissant le bit **SYSRESETREQ** du registre **AIRCR**)
- Reset sur condition **Standby** et/ou **Stop**

Le registre **RCC\_CSR** est le seul registre qui n'est pas remis dans son état initial par un reset. Il contient un champ de bits permettant de déterminer quelle a été la cause du reset.

Il est courant sur une application d'effectuer un contrôle du mode de reset :

- Si le reset vient d'une cause externe (mise sous tension), un compteur est mis à 0 et mémorisé en mémoire flash.
- Si le reset vient d'un watchdog, ce compteur est incrémenté et mémorisé en mémoire flash.
  - Si le compteur dépasse un certain seuil, l'application passe en mode dégradé, en réduisant sa consommation, en indiquant son état d'erreur (led rouge par exemple).
  - Il sera alors nécessaire d'effectuer un reset externe pour redémarrer.

On peut coupler ce contrôle avec une surveillance sur les moments des reset et décider de passer l'application en mode dégradé si un nombre excessif de reset a eu lieu durant une période de temps.

#### 1.6.4.2 Horloge système

L'horloge principale est l'horloge système (**SYSCLK**).

Il est possible de choisir la source de **SYSCLK**, il y a trois sources possibles :

**HSI** High Speed Internal : 16 MHz RC

**HSE** High Speed External : qui peut être

- Clock externe
- Quartz externe

**Main PLL** : Prend comme entrée HSI ou HSE

- Une sortie High Speed **SYSCLK** (max 84 MHz)
- Une sortie clock pour l'USB à 48 MHz, le générateur de nombres aléatoires et le SDIO.

Il y a deux sources secondaires pour des périphériques particuliers :

**LSI RC** Low Speed Internal : 32 kHz pour le Independent Watchdog, auto-wakeup.

**LSE crystal** Low Speed External : 32.768 kHz pour le RTCCLK.

Lors du Reset :

- HSI est sélectionné
- Le changement vers HSE ou PLL ne se fait que si ils sont présents et stabilisés.
- Le registre **RCC\_CR** (Reset and Clock Control Control Register) indique l'état de la source en cours.
- Différents mécanismes de protection en cas de perte de HSE (**WWDG**).
- Et quelques contraintes qui protègent :
  - On ne peut pas arrêter une horloge utilisée (Le **LSI** et le **IWDG**).

Le microcontrôleur peut aussi être une source d'horloge, il y a deux sorties horloges configurables :

**MC01** Au choix parmi

- HSI
- HSE
- LSE
- PLL

**MC02** • HSE  
• PLL  
• System Clock **SYSCLK**  
• **PLLI2S**

Ce sont des fonctions spéciales, les **GPIO** correspondantes doivent être configurées en **AF** (Alternate Function).

Cela peut être utilisé pour contrôler depuis l'extérieur les fréquences internes de fonctionnement. Il est aussi possible de faire des mesures internes pour calibrer les horloges.

La figure 1.18 montre l'arbre complet des horloges. Il est possible de jouer avec les 'prescaler' pour arriver à la fréquence désirée pour les périphériques et pour le cœur. La fréquence devra être adaptée à l'application, cela joue sur les performances et sur la consommation.



Avec notre carte Nucleo :

- OSC\_IN, HSE = 8 MHz
- AHB Prescaler : 1
- APB1 Prescaler : 2
- APB2 Prescaler : 1
- PLLM : 25
- PLLN : 336
- PLLP : 4
- PLLQ : 7

$$\begin{aligned}f_{(VCOclock)} &= f_{(PLLclockinput)} * (PLLN/PLLM) \\f_{(PLLgeneralclockoutput)} &= f_{(VCOclock)}/PLLP \\f_{(VCOclock)} &= 8000000 * 336/25 = 107520000 \\f_{(PLLgeneralclockoutput)} &= 107520000/4 = 26880000\end{aligned}$$

Les formules 'magiques' viennent de la documentation [8] (p104).

⇒ **SYSCLK = 26 880 000 Hz**

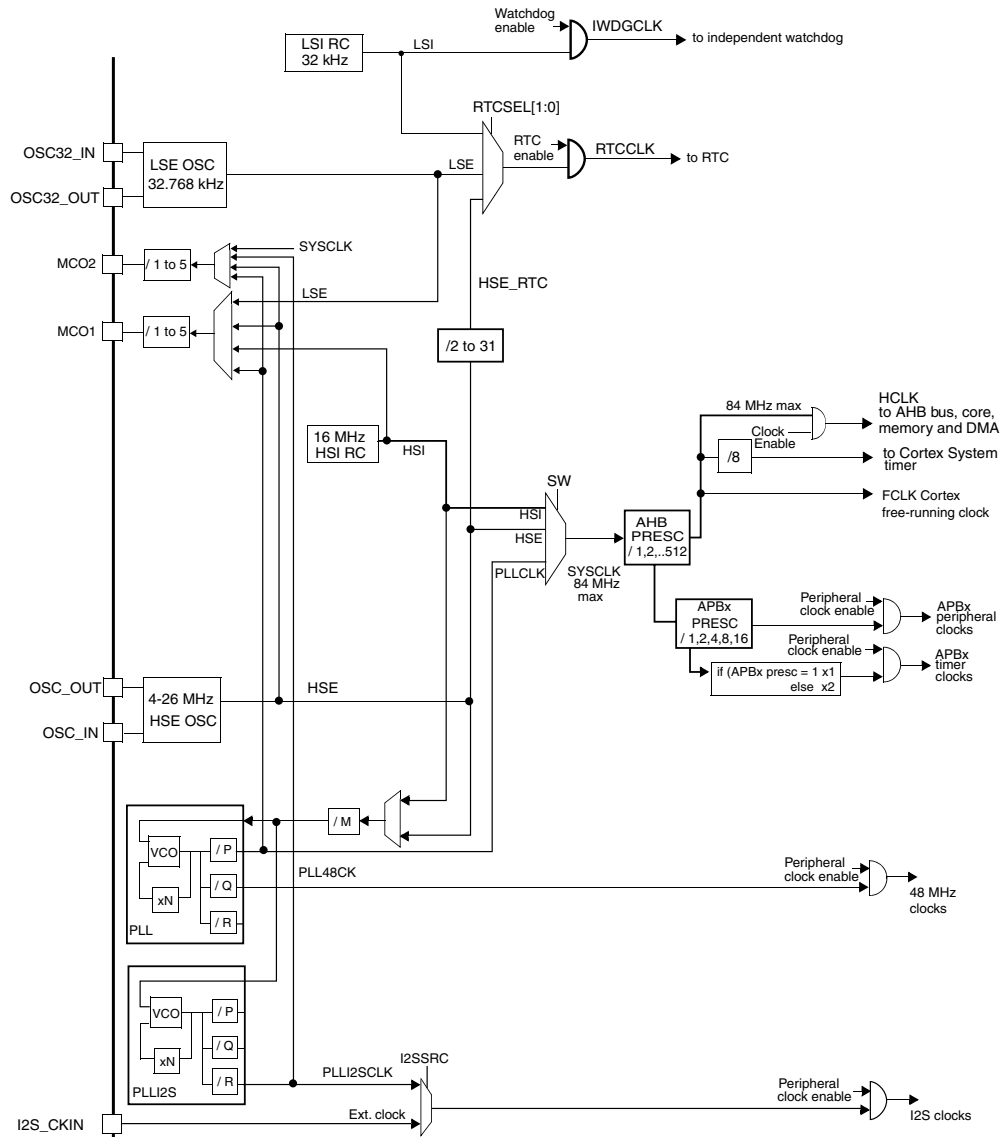


FIGURE 1.18 – Arbre des horloges

### 1.6.4.3 Horloge des périphériques

Lors du reset, les horloges des périphériques sont désactivées. Tant qu'une horloge ne sera pas activée pour un périphérique, il ne sera pas possible de le configurer ou de l'utiliser.

Si un périphérique n'est plus utilisé, il est bon de désactiver son horloge pour réduire la consommation globale du système.

Les registres `RCC*ENR` permettent de définir individuellement si une horloge est activée ou non.

La figure 1.19 montre le registre de configuration des horloges dédiées au bus `AHB1`, qui contient entre autres les horloges des `GPIO`. Il y aura l'équivalent pour les autres bus (`AHB1`, `AHB2`, `APB1`, `APB2`). Il existe des registres similaires pour les horloges en mode basse consommation.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									DMA2EN	DMA1EN	Reserved				
									rw	rw					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN		Reserved			GPIOH EN	Reserved		GPIOEEN	GPIOD EN	GPIOC EN	GPIOB EN	GPIOA EN
			rw					rw			rw	rw	rw	rw	

FIGURE 1.19 – Registre de contrôle des horloges

Le listing 1.2 montre l'autorisation de l'horloge des périphériques `GPIOA` et `GPIOB`. Il faudra activer ces horloges avant toute utilisation du périphérique correspondant.

```
RCC->AHB1ENR |= 3; // activation de l'horloge des GPIOA et GPIOB
```

Listing 1.2 – Activation de l'horloge pour des périphériques

## 1.6.5 GPIO

La définition de tous les registres relatifs aux `GPIO` se trouve en [8] (chapitre 8.4).

Le nombre de ports de `GPIO` dépend du modèle de `STM32F401` retenu.

Chaque port `GPIOA`, `GPIOB`, `GPIOC`, ... permet de configurer 16 entrées/sorties. On pourra utiliser les notations `GPIOA.0` à `GPIOA.15` pour les bits du `GPIOA`.

Des registres permettent de configurer pour chaque entrée/sortie :

- Si elle est configurée en entrée (flottante, pull-up/down, analogique) ou en sortie (push-pull, open drain, pull-up/down), la vitesse des sorties est aussi configurable
- Si la sortie est pilotée (forcée) par un registre (`GPIOx_ODR`, `GPIOx_BSRRH`, `GPIOx_BSSRL`) ou comme un périphérique par une fonction alternative (`SPI`, `I2C`, `UART`, ...)
- Si l'entrée est relue par un registre (`GPIOx_IDR`) ou comme un périphérique par une fonction alternative.

- ...

Chaque entrée/sortie peut être individuellement configurée (voir figure 1.20).

#### 1.6.5.1 Les registres relatifs aux GPIO

- Chaque port GPIO adresse 16 broches
- Chaque broche peut être connectée à une des fonctions alternatives
- Les fonctions BSSR permettent un accès atomique aux broches
- Chaque broche peut être configurée individuellement
  - En entrée, sortie, analogique, ou fonction alternative
  - En push-pull ou en open-drain
  - Avec pull-up, pull-down ou sans
  - Pour son slew-rate (la vitesse de bascule)

@0x00 MODER MOde REgister ⇒ entrée ou sortie

@0x04 OTyPER Output TYPE Register ⇒ push-pull ou open-drain

@0x08 OSPEEDR Output SPEED Register ⇒ low, medium, fast, high

@0x0C PUPDR Pull-Up/Pull-Down Register ⇒ pull-up, pull-down ?

@0x10 IDR Input Data Register ⇒ lecture de valeurs

@0x14 ODR Output Data Register ⇒ écriture de valeurs

@0x18 BSSR Bit Set/Reset Register ⇒ écriture par bits

@0x1C LCKR configuration LoCK Register ⇒ protection de configurations

@0x20 AFRL Alternate Function Low Register ⇒ fonctions alternatives

@0x24 AFRH Alternate Function High Register ⇒ fonctions alternatives  
(suite)

En A.6 se trouve un résumé des registres relatifs aux GPIO.



```

1 void HardwareInit(void)
2 {
3     // Led2 sur PA5
4     RCC->AHB1ENR |= 1; // GPIOA clock enable
5     // PA5: output b11=0 b10=1
6     GPIOA->MODER &= 0b11111111111111111111111111111111;
7     GPIOA->MODER |= 0b000000000000000000000000100000000000;
8     // PA5: pushpull b5 = 0
9     GPIOA->OTYPER &= 0b11111111111111111111111111111111;
10    // PA5: speed high b11=1 b10=1
11    GPIOA->OSPEEDR |= 0b000000000000000000000000110000000000;
12    // PA5: no pull-up, no pull-down b11=0 b10=0
13    GPIOA->PUPDR &= 0b11111111111111111111111111111111;
14    // PA5: output 0 (au départ), b5=0
15    GPIOA->ODR &= 0b11111111111111111111111111111111; // on pourrait aussi
    ↪ utiliser BSRRH
16
17    // Bouton poussoir B1 sur PC13
18    RCC->AHB1ENR |= 4; // GPIOC clock enable
19    // PC13: input b27=0 b26=0
20    GPIOA->MODER &= 0b11110011111111111111111111111111;
21    // PC13: no pull-up, no pull-down b27=0 b26=0
22    GPIOA->PUPDR &= 0b11110011111111111111111111111111;
23 }
24 int main(void)
25 {
26     SystemInit();
27     HardwareInit();
28     for(;;)
29     {
30         if (0 == (GPIOC->IDR & 0b0010000000000000))
31         {
32             if (GPIOA->ODR & (1<<5))
33             {
34                 GPIOA->BSRRH = 1<<5;
35             }
36             else
37             {
38                 GPIOA->BSRRL = 1<<5;
39             }
40         }
41     }
42 }

```

Listing 1.4 – GPIO, led et bouton poussoir

### 1.6.6 Contrôleur DMA

Le DMA permet d'effectuer des transferts de données de :

- la mémoire vers la mémoire
- un périphérique vers la mémoire
- la mémoire vers un périphérique

Durant un transfert **DMA**, il n'y a pas d'actions du microcontrôleur. Cela permet de faire effectuer d'autres tâches par le microcontrôleur (amélioration des performances du système), ou de le mettre en veille (baisse de la consommation du système).

Il y a 2 contrôleurs **DMA**, gérant chacun 8 flux, chaque flux peut gérer 8 requêtes.

Les contrôleurs arbitrent les transferts, en gérant des priorités sur les données transférés. La configuration des transferts comprend en particuliers :

- Le nombre de données à transférer (de 1 à 65535)
- L'incrémentation ou non des adresses de la source et de la destination.
- Une gestion **FIFO** ou directe avec des seuils d'alerte de remplissage.
- La configuration en buffer circulaire, et le double buffering pour les transferts en continu.

Le listing 1.5 montre une utilisation du **DMA** pour transférer des données de la mémoire vers la mémoire. La fonction `memcpy` a une vitesse de transfert plus importante, mais le **DMA** à l'avantage de libérer le microcontrôleur.

```

1  #define DATA_COUNT 850
2  uint32_t uSource[DATA_COUNT];
3  uint32_t uDestination[DATA_COUNT];
4
5  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE); // clock du DMA2
6
7  // configuration du transfert DMA
8  DMA_Cmd(DMA_Channel_0, DISABLE);
9  DMA_DeInit(DMA2_Stream0);
10 DMA_InitTypeDef dma_m2m;
11
12 dma_m2m.DMA_Channel = DMA_Channel_0;
13 dma_m2m.DMA_Memory0BaseAddr = (uint32_t)uDestination;
14 dma_m2m.DMA_PeripheralBaseAddr = (uint32_t)uSource;
15 dma_m2m.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
16 dma_m2m.DMA_PeripheralDataSize = DMA_MemoryDataSize_Word;
17 dma_m2m.DMA_DIR = DMA_DIR_MemoryToMemory;
18 dma_m2m.DMA_Mode = DMA_Mode_Normal;
19 dma_m2m.DMA_MemoryInc = DMA_MemoryInc_Enable;
20 dma_m2m.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
21 dma_m2m.DMA_BufferSize = sizeof(uSource);
22 dma_m2m.DMA_Priority = DMA_Priority_High;
23 dma_m2m.DMA_MemoryBurst = DMA_MemoryBurst_Single;
24 dma_m2m.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
25 dma_m2m.DMA_FIFOMode = DMA_FIFOMode_Enable;
26 dma_m2m.DMA_FIFOThreshold = DMA_FIFOStatus_Full;
27
28 DMA_Init(DMA2_Stream0, &dma_m2m);
29
30 // remplissage (pour tests)
31 int i;
32 for(i=0; i<DATA_COUNT; i++)
33 {
34     uSource[i] = 100+i;
35     uDestination[i] = 555;
36 }
37
38 DMA_Cmd(DMA2_Stream0, ENABLE); // début de la transaction
39 while (DMA_GetFlagStatus(DMA2_Stream0, DMA_FLAG_TCIF0) == RESET); // attente
40
41 // maintenant uDestination contient une copie de uSource

```

Listing 1.5 – Utilisation du DMA

### 1.6.7 Interruptions et évènements

Les interruptions sont gérées par le NVIC. Elles sont exécutées de manière asynchrone par rapport à l'exécution du programme et provoquent un branchement vers une adresse configurée dans la table des vecteurs d'interruptions.



Il y a un grand nombre de sources d'interruptions, qui peuvent provenir des périphériques (GPIO, USART, ...) ou être générées par des dysfonctionnement du système.

En reprenant l'exemple 1.5, on peut ajouter une interruption qui sera appelée quand le transfert DMA sera terminé. Ce nouveau code se trouve sur le listing 1.7. Dans ce nouvel exemple, il y a une attente sur une variable `irqTransferCompleteDone`, qui est activée par la routine d'interruption du listing 1.6. Dans une véritable application, il faudrait plutôt endormir le microcontrôleur.

```
1  volatile uint32_t irqTransferCompleteDone = 0;
2  void DMA2_Stream0_IRQHandler(void)
3  {
4      //Test DMA2 Channel1 Transfer Complete interrupt
5      if(DMA_GetITStatus(DMA2_Stream0, DMA_IT_TCIF0))
6      {
7          irqTransferCompleteDone = 1;
8          DMA_ClearITPendingBit(DMA2_Stream0, DMA_FLAG_TCIF0);
9      }
10 }
```

Listing 1.6 – Routine d'interruption du DMA

```

1  #define DATA_COUNT 850
2  uint32_t uSource[DATA_COUNT];
3  uint32_t uDestination[DATA_COUNT];
4
5  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE); // clock du DMA2
6
7  // configuration du transfert DMA
8  DMA_Cmd(DMA_Channel_0, DISABLE);
9  DMA_DeInit(DMA2_Stream0);
10 DMA_InitTypeDef dma_m2m;
11
12 dma_m2m.DMA_Channel = DMA_Channel_0;
13 dma_m2m.DMA_Memory0BaseAddr = (uint32_t)uDestination;
14 dma_m2m.DMA_PeripheralBaseAddr = (uint32_t)uSource;
15 dma_m2m.DMA_MemoryDataSize = DMA_MemoryDataSize_Word;
16 dma_m2m.DMA_PeripheralDataSize = DMA_MemoryDataSize_Word;
17 dma_m2m.DMA_DIR = DMA_DIR_MemoryToMemory;
18 dma_m2m.DMA_Mode = DMA_Mode_Normal;
19 dma_m2m.DMA_MemoryInc = DMA_MemoryInc_Enable;
20 dma_m2m.DMA_PeripheralInc = DMA_PeripheralInc_Enable;
21 dma_m2m.DMA_BufferSize = sizeof(uSource);
22 dma_m2m.DMA_Priority = DMA_Priority_High;
23 dma_m2m.DMA_MemoryBurst = DMA_MemoryBurst_Single;
24 dma_m2m.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
25 dma_m2m.DMA_FIFOMode = DMA_FIFOMode_Enable;
26 dma_m2m.DMA_FIFOThreshold = DMA_FIFOStatus_Full;
27
28 DMA_Init(DMA2_Stream0, &dma_m2m);
29
30 // Enable DMA2 Channel Transfer Complete interrupt
31 DMA_ITConfig(DMA2_Stream0, DMA_IT_TC, ENABLE);
32
33 // configuration de l'interruption
34 NVIC_InitTypeDef NVIC_InitStructure;
35 //Enable DMA2 channel IRQ Channel */
36 NVIC_InitStructure.NVIC_IRQChannel = DMA2_Stream0_IRQn;
37 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
38 NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
39 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
40 NVIC_Init(&NVIC_InitStructure);
41 NVIC_EnableIRQ(DMA2_Stream0_IRQn);
42
43 // remplissage (pour tests)
44 int i;
45 for(i=0; i<DATA_COUNT; i++)
46 {
47     uSource[i] = 100+i;
48     uDestination[i] = 555;
49 }
50
51 irqTransferCompleteDone = 0;
52 DMA_Cmd(DMA2_Stream0, ENABLE);
53 while(irqTransferCompleteDone == 0); //attente
54
55 // maintenant uDestination contient une copie de uSource

```

Listing 1.7 – Utilisation et configuration d'une interruption avec le DMA

### 1.6.7.1 Le SysTick

Une interruption particulière va être utilisée, elle est nommée **SysTick** et permet d'implémenter simplement des actions périodiques. Cela sera particulièrement utile lors de l'utilisation d'un système d'exploitation temps réel, cette interruption va servir à cadencer l'appel de l'ordonnanceur du système.

La fréquence de cette interruption peut être configurée.



#### Handler d'interruption **SysTick\_Handler**

Dès que le **SysTick** va être configuré une interruption peut se produire. Vous devez donc fournir un gestionnaire d'interruption dédié (une fonction **SysTick\_Handler**).

Le listing 1.8 montre la configuration des registres relatifs au **SysTick**. La documentation complète se trouve dans le manuel de programmation [10] au chapitre 4.5.

On peut remarquer :

- La formule de calcul permettant de charger le registre **LOAD** avec la valeur nécessaire. Cette valeur dépend de la période du **SysTick** désirée, mais aussi de la source d'horloge (**AHB/8** ou **Processor clock AHB**).
- Le registre **CTRL** permet de définir la source d'horloge. Mais aussi le fait de générer ou pas une interruption.

Dans notre exemple, nous allons utiliser l'horloge processeur (ici 26.88 MHz), et configurer les registres pour avoir :

- Un tick toutes les ms
- Une interruption quand le **SysTick** se déclenche. Le listing 1.9 montre un gestionnaire d'interruption simple, qui compte juste les millisecondes écoulées.

```

1 void TimerInit(void)
2 {
3     // SYSCLK = 26880000Hz => 26880-1 pour les ticks /ms
4     SysTick->LOAD = 0x0FFFFFFF & (uint32_t)(26880 - 1UL);
5
6     // RAZ
7     SysTick->VAL = 0UL;
8
9     // Let's go...
10    SysTick->CTRL = 0b111;
11 }

```

Listing 1.8 – Configuration du SysTick

```

1 volatile uint64_t ulTicksMs = 0;
2 void SysTick_Handler(void)
3 {
4     ulTicksMs++; // pour utilisation externe
5 }

```

Listing 1.9 – Gestion d'interruption du SysTick

L'utilisation de la `StdPeriph` permet de simplifier l'écriture, en encapsulant les appels aux registres. Le listing 1.10 est équivalent au listing 1.8.

```

1 void TimerInit(void)
2 {
3     SystemCoreClockUpdate(); // Mise à jour de la variable globale SystemCoreClock
4     SysTick_Config(SystemCoreClock/1000); // SysTick interrupt set to 1ms
5 }

```

Listing 1.10 – Configuration du SysTick

### 1.6.7.2 Les interruptions extérieures

Les interruptions extérieures EXTI (au nombre de 23) sont déclenchées par des actions sur des GPIO :

- EXTI0 pour les entrées GPIOA.0, GPIOB.0, GPIOC.0, ...
- EXTI1 pour les entrées GPIOA.1, GPIOB.1, GPIOC.1, ...
- EXTI15 pour les entrées GPIOA.15, GPIOB.15, GPIOC.15, ...

Les autres interruptions EXTI sont connectées à des périphériques (PVD, RTC, USB OTG, ...).

Les interruptions peuvent être aussi déclenchées par logiciel (Software Interrupt).

### 1.6.8 Autres périphériques

L'utilisation des autres périphériques sera abordée par des exemples dans les TD.

### 1.6.9 Identification

#### 1.6.9.1 Unique device ID

En relisant simplement une adresse du STM32F401, on peut récupérer son identifiant unique.

Cela peut servir à :

- Discriminer entre 2 appareils (utilisation comme numéro de série)
- Des fonctions de sécurité, ou de vérification d'intégrité.

Le listing 1.11 montre comment récupérer l'identifiant unique.

```
#define ID_FLASH_ADDRESS 0xFFFF7A22
#define ID_UNIQUE_ADDRESS 0xFFFF7A10

#define TM_ID_GetUnique32(x) ((x >= 0 && x < 3) ? (*(uint32_t *)
↪ (ID_UNIQUE_ADDRESS + 4 * (x))) : 0)

//...
printf("Unique:%08x%08x%08x\n", TM_ID_GetUnique32(0), TM_ID_GetUnique32(1),
↪ TM_ID_GetUnique32(2));
//...
```

Listing 1.11 – Lecture de l'identifiant unique

#### 1.6.9.2 Taille de la mémoire flash

Il est possible de relire la mémoire flash disponible sur le microcontrôleur. Le listing 1.12 montre comment réaliser cette opération. Le résultat est en kB.

```
#define ID_FLASH_ADDRESS 0x1FFF7A22
#define TM_ID_GetFlashSize() (*(uint16_t *) (ID_FLASH_ADDRESS))

//...
printf("Flash:%u kiB\n", (unsigned)TM_ID_GetFlashSize());
//...
```

Listing 1.12 – Lecture de la taille de la mémoire flash

# 2

## Du hardware au software

*Ce chapitre traite de la programmation en assembleur et des principes de l'architecture ARM.*

*“The question of whether Machines Can Think... is about as relevant as the question of whether Submarines Can Swim.”*

- Edsger Dijkstra,

### 2.1 De l'analogique au numérique

Les grandeurs physiques habituelles (tension, température, luminosité, ...) sont des grandeurs analogiques. Et nos sens humains sont aussi analogiques, pourtant il est plus simple de traiter l'information de manière numérique par des valeurs discrètes (des nombres).

#### 2.1.1 Du bit aux octets

Un système à 2 états représente la plus petite unité d'information (haut/bas, vrai/faux, ...). L'unité associée est le bit <sup>1</sup>. Un état est représenté par le chiffre 0, et l'autre par le chiffre 1.

Pour décrire une information plus complexe, on regroupe les bits :

- Par 4 : un quartet <sup>2</sup>
- Par 8 : un octet <sup>3</sup>
- Par 16 : un mot <sup>4</sup>

---

1. BInary digiT en anglais

2. nibble en anglais

3. byte en anglais

4. word en anglais

### 2.1.2 Codage de l'information

- Avec 1 bit, il est possible de coder 2 états (0 ou 1).
- Avec 4 bits, il sera possible d'en coder 16 (de 0 à 15).
- Avec 8 bits, il sera possible d'en coder 256 (de 0 à 255).
- Avec  $n$  bits, il sera possible de coder  $2^n$  états.

On représente habituellement les bits de 0 à  $n - 1$ , de droite à gauche. Le bit le plus à droite est appelé le LSB (Least Significant Bit),  $b_0$ . Le bit le plus à gauche est appelé le MSB (Most Significant Bit),  $b_7$  pour un octet,  $b_{15}$  pour un word.

La table 2.2 donne les valeurs de quelques octets. Ce que représente individuellement chaque bit dépend de l'application.

Il n'y a pas ici de notion de signes. Toutes les valeurs représentées ici sont positives. Le codage des nombres entiers signés (complément à 2) ne sera pas abordé pour l'instant.

### 2.1.3 Du binaire à l'hexadécimal

On travaillera souvent avec des valeurs hexadécimales, qui permettent de coder de manière simple des nombres.

Chaque quartet va avoir une valeur hexadécimale. Un quartet contient 4 bits, il y aura donc 16 groupes différents de 4 bits. Les chiffres 0 à 9 vont être utilisés, ainsi que les 6 premières lettres de l'alphabet A à F. La table 2.1 contient les correspondances.

Le nombre binaire 00100011, sera donc noté 23 en hexadécimal. Pour éviter toute ambiguïté, les nombres hexadécimaux seront précédés de 0x, et les nombres binaires de 0b.

$$0b00100011 = 0x23 = 35$$

0000	0001	0010	0011	0100	0101	0110	0111
0	1	2	3	4	5	6	7
1000	1001	1010	1011	1100	1101	1110	1111
8	9	A	B	C	D	E	F

TABLE 2.1 – Nom des quartets



Valeur	$2^7$ $b_7$	$2^6$ $b_6$	$2^5$ $b_5$	$2^4$ $b_4$	$2^3$ $b_3$	$2^2$ $b_2$	$2^1$ $b_1$	$2^0$ $b_0$
255 0xFF	1	1	1	1	1	1	1	1
252 0xFC	1	1	1	1	1	1	0	0
35 0x23	0	0	1	0	0	0	1	1
0 0x00	0	0	0	0	0	0	0	0
201 0xC9	1	0	0	0	0	0	0	1
128 0x80	1	0	0	0	0	0	0	0
127 0x7F	0	1	1	1	1	1	1	1

TABLE 2.2 – Représentation mémoire de quelques octets

## 2.2 L'architecture du point de vue du développeur

Le microcontrôleur sera défini par le jeu d'instructions qu'il va supporter (on retrouve les familles, ARM, x86, MIPS, ...) et les opérandes sur lesquels vont s'appliquer ces instructions.

- les registres
- la mémoire
- des constantes

La première étape est l'apprentissage du langage du microcontrôleur. Les mots du langage sont appelés des instructions. Tous les programmes qui vont s'exécuter sur un même microcontrôleur vont utiliser le même ensemble d'instructions (on parlera de jeu d'instructions<sup>5</sup>).

Sur un microprocesseur (sur un PC par exemple), les applications les plus complexes (traitement de texte, code de calcul, logiciel de simulation, ...) sont compilées en une suite d'instructions simples (élémentaires) :

- additionner
- soustraire
- effectuer un branchement
- ...

Les instructions sur le microprocesseur (microcontrôleur) comporte à la fois le type d'opération à exécuter et les opérandes sur lesquels les appliquer (comme vu précédemment : la mémoire, les registres, ...).

Les microcontrôleurs comprennent uniquement les 0 et les 1, les instructions seront encodées en nombres binaires appelé langage machine.

---

5. instruction set en anglais

Sur un ARM les instructions sont encodées sur 32 bits, mais pour un humain (normal!) il est difficile de lire une suite d'instructions telle que :

- 00011000 10010000 10001010 00010001
- 00011101 11101111 11100010 00010011
- etc...

L'assembleur est la représentation humainement compréhensible du langage machine, chaque instruction d'assembleur (comme en langage machine) va contenir l'opération, puis le ou les opérandes.

## 2.2.1 Quelques instructions

Nous avons vu en langage C des instructions du genre :

```
compteur = valeur + increment;
```

Ce qui donne en assembleur :

```
ADD a, b, c
```

ADD est la mnémonique (l'instruction à exécuter), b et c sont les opérandes sources, a est l'opérande de destination. Les opérandes a,b,c représentent les zones mémoires qui stockent les valeurs des paramètres.

De la même manière :

```
compteur = valeur - increment;
```

Donnerait en assembleur

```
SUB a, b, c
```

Cela correspond à un des principes qui conduit à la réalisation des architectures ARM : la simplicité; des instructions similaires sont codées de manières similaires.



### Principe 1

| La simplicité

Des opérations plus complexes en C    Entraîneront la génération des codes  
comme    suivants

```
compteur = valeur + increment -  
↪ offset;
```

```
ADD t,b,c    ; t = b + c  
SUB a,t,d    ; a = t - d
```

En assembleur, les commentaires se situent après le caractère ;.

C'est à dire, une suite d'instructions simples pour des opérations complexes.



### Principe 2

| Optimiser les cas les plus courants

Ceci a été réalisé chez ARM en utilisant uniquement un jeu d'instructions simples et communes. Le nombre d'instructions a été réduit au minimum, pour permettre un décodage instructions/opérandes rapide et simple. Les opérations plus complexes (mais aussi plus rares) sont réalisées avec plusieurs instructions élémentaires. L'architecture ARM est une architecture RISC<sup>6</sup>, à l'opposé des architecture x86 qui sont des CISC<sup>7</sup>, comprenant un grand nombre d'instructions complexes.

La taille des instructions étant fixes pour un processeur, pour coder 64 instructions : 6 bits suffisent, pour en coder 256, il faut 8 bits. Dans les processeurs CISC, l'existence des instructions les plus rares fait que toutes les instructions doivent être codées avec le plus grand nombre de bits.

Les valeurs des opérandes seront stockés :

- En mémoire
- Dans des registres
- Directement dans l'instruction (valeur immédiate constante)

Les opérandes stockés dans des registres ou dans l'instruction (constante) seront accédés très rapidement, mais auront une taille limitée.

- 32 bits pour les registres
- 8, 12, 16, 32 bits pour les valeurs immédiates (suivant les instructions)

Les autres données provenant de la mémoire sont accessibles plus lentement, les données sur un ARM ont 32 bits.

### 2.2.2 Les registres

Les instructions ont besoin de récupérer les opérandes rapidement. Mais les opérandes stockés en mémoire sont lents à récupérer, la plupart des architectures définissent un certain nombre de registres qui contiennent des opérandes.

L'architecture ARM utilise 16 registres. Moins il y a de registres, plus il est rapide d'y accéder.

---

6. Reduced Instruction Set Computing

7. Complex Instruction Set Computing

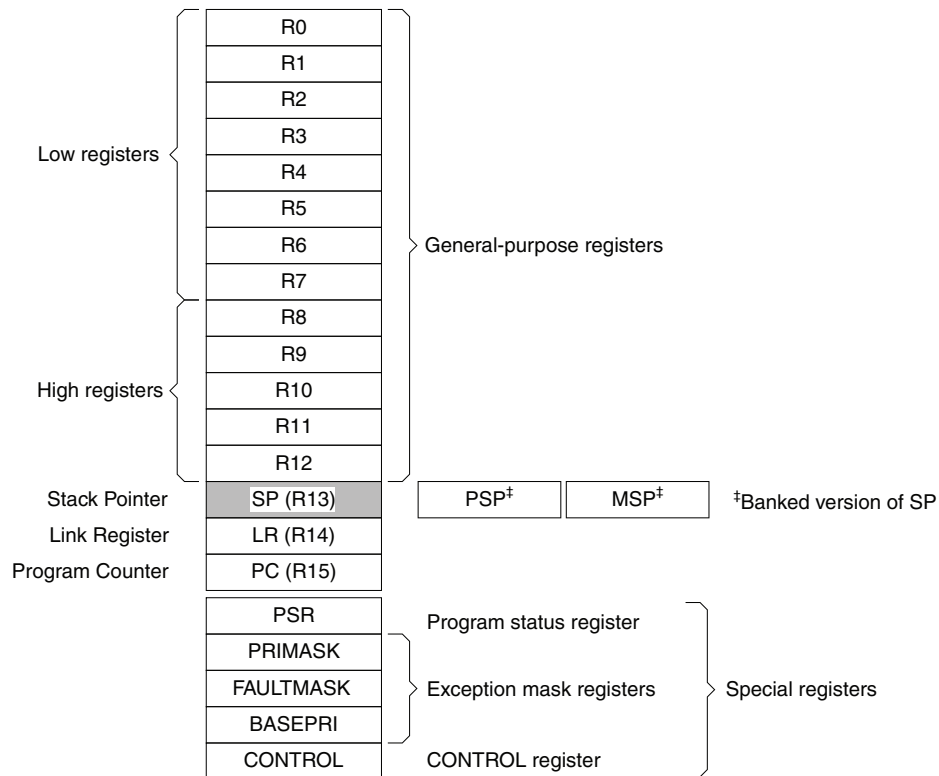


FIGURE 2.1 – Registres du STM32F401xD/xE



### Principe 3

| Plus petit, plus rapide

Par exemple rechercher une information dans un petit carnet est plus rapide (normalement) que de chercher l'information dans une bibliothèque.

La figure 2.1 montre les 16 registres utilisés par un ARM et les registres d'états et de contrôles. Les registres R0 à R12 sont utilisés pour stocker des variables. Les registres R13 à R15 ont des noms particuliers correspondant à des fonctionnalités spéciales, des registres supplémentaires servent à stocker l'état et la configuration du microcontrôleur.

**R0-R12** Pas de fonctions particulières, ce sont des registres généraux qui sont utilisés comme variables pour les opérations. Mais les registres R0-R3 ont aussi des significations particulières lors des appels de fonctions :

- R0 Paramètre/Valeur retournée/Variable temporaire
- R1-R3 Paramètres, variables temporaires

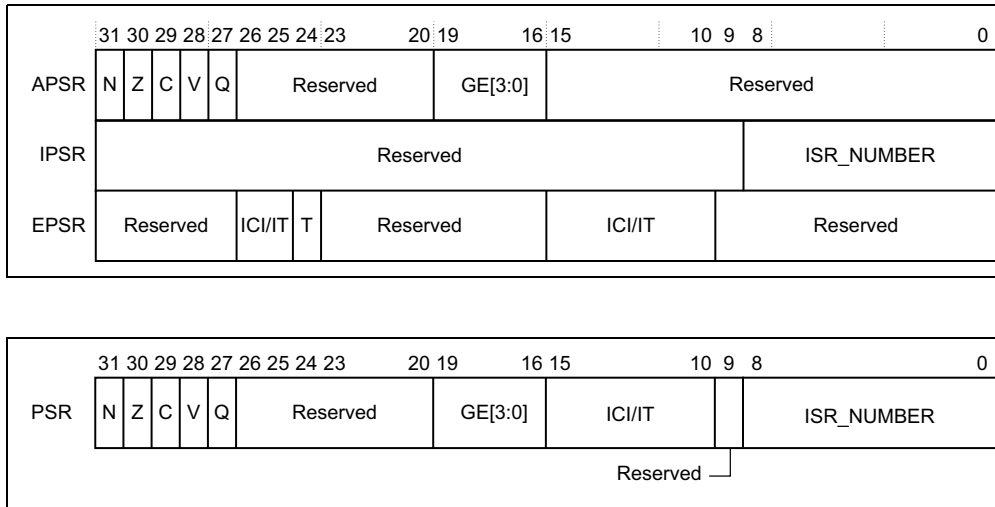


FIGURE 2.2 – Registres d'états du STM32F401xD/xE

**MSP Main Stack Pointer** le pointeur de pile principal

**PSP Process Stack Pointer** le pointeur de pile de processus

**LR Link Register** sert à stocker l'adresse de retour des fonctions, exceptions, ...

**PC Program Counter** contient l'adresse de l'instruction courante, initialisé à 0x00000004 lors du reset.

**PSR Program Status Register** contient une combinaison de 3 registres d'état APSR, IPSR, EPSR. Ces trois registres sont des champs de bits mutuellement exclusif du PSR, voir la figure 2.2.

- **APSR Application Program Status Register** contient l'état et les conditions de la dernière instructions exécutées.
- **IPSR Interrupt Program Status Register** contient le type d'interruption en cours.
- **EPSR Execution Program Status Register** contient l'état d'exécution des instructions. En particuliers le mode thumb ou non.

**PRIMASK** contient un bit de protection contre les exceptions suivant une priorité configurable (dans BASEPRI).

**FAULTMASK** contient un bit de protection contre les exceptions.

**BASEPRI** contient la priorité d'exécution, et permet de se protéger des exceptions de priorités inférieures.

**CONTROL** contient des bits indiquant le niveau d'exécution des instructions, l'état du coprocesseur, ...

En C

```
compteur = compteur + 3;
resultat = compteur - 10;
```

En assembleur

```
; avec compteur en R7 et resultat en
↪ R8
ADD R7, R7, #3
SUB R8, R7, #0xA
```

Listing 2.1 – Exemple de calculs avec valeurs immédiates

### 2.2.3 Valeurs constantes, valeurs immédiates

En plus des opérandes registres, les instructions ARM peuvent utiliser des constantes (appelées valeurs immédiates). Elles sont appelées immédiates car il n'y a pas d'accès supplémentaires à faire en mémoire pour accéder à la valeur.

Le listing 2.1 montre des calculs effectués avec des valeurs immédiates.

En C

```
compteur = 0;
resultat = 2660;
```

En assembleur

```
; avec compteur en R7 et resultat en
↪ R8
MOV R7, #0
MOV R8, #0xA64
```

Listing 2.2 – Exemple d'affectation avec des valeurs immédiates

Le listing 2.2 montre l'initialisation de variables avec des valeurs immédiates (0 et 2660=0xA64) avec l'instruction **MOV**.

Dans l'architecture ARM, les instructions travaillent uniquement sur des valeurs immédiates ou sur des registres. Il faut donc des instructions pour transférer les données depuis la mémoire vers un registre, ou depuis un registre vers la mémoire. Dans le cas contraire nous serions limité à 15 variables (le 16<sup>ème</sup> registre est le **Program Counter**).

L'architecture ARM utilise un espace mémoire de 32 bits, contenant des données sur 32 bits. Soit au maximum  $2^{32} = 4\,294\,967\,296$  octets. Les mots de données de 32 bits (4 octets de 8 bits) sont donc sur des adresses qui sont des multiples de 4. Pour des raisons d'efficacité, les mots données de 32 bits sont toujours alignés sur des frontières de 4 octets.

L'instruction Load Register **LDR**, permet de lire un mot depuis une adresse mémoire et de le stocker dans un registre. Il y a plusieurs méthodes d'accès à la mémoire (avec des offsets, des calculs, ...)

L’instruction Store Register **STR** permet de stocker le contenu d’un registre dans une zone mémoire. Là aussi, il y a plusieurs modes de calculs de l’adresse finale en fonction des paramètres.

### 2.2.4 Little Endian, Big Endian

La mémoire peut être organisée en mode Big Endian ou Little Endian (grand boutisme et petit boutisme). Dans les architectures Big Endian, l’octet 0 est sur le MSB (Most significant Bit), dans les architectures Little Endian, l’octet 0 est sur le LSB (Least Significant Bit). Les architectures ARM utilisent en général du Little Endian, bien qu’il soit possible de fonctionner en Big Endian. Il n’y a normalement pas de choix à faire, et cela est même transparent pour le développeur, tant qu’il n’échange pas de données binaires avec une autre architecture.

...	...	...	...	...	...	...	...	
C	D	E	F	0xC	F	E	D	C
8	9	A	B	0x8	B	A	9	8
4	5	6	7	0x4	7	6	5	4
0	1	2	3	0x0	3	2	1	0
MSB			LSB	Adr	MSB			LSB

TABLE 2.3 – Big-Endian et Little-Endian

## 2.3 Les instructions assembleurs

Les instructions assembleurs sont en nombre limité et dépendent du jeu d’instructions utilisé. Dans le document [1] on peut trouver une définition des jeux d’instructions.

### 2.3.1 Les instructions logiques

Instructions bits à bits.

En assembleur

- AND
- ORR (OR)
- EOR (XOR)
- BIC (Bit Clear)
- MVN (MoVe and Not)

En C

- `&` : opérateur **et**
- `|` : opérateur **ou**
- `^` : opérateur **ou exclusif**
- Similaire à `& ~(1 << n)`
- `~`

### 2.3.2 Les instructions de décalage

En assembleur	En C
<ul style="list-style-type: none"> <li>• LSL Logical Shift Left</li> <li>• LSR Logical Shift Right (remplissage avec des 0)</li> <li>• ASR Arithmetic Shift Right (remplissage avec le signe)</li> <li>• ROR ROTate Right il n'y a pas de ROL (c'est pareil!)</li> </ul>	<ul style="list-style-type: none"> <li>• « : décalage à gauche d'un nombre de bits (ce qui revient à faire une multiplication par une puissance de 2)</li> <li>• » : décalage à droite d'un nombre de bits (ce qui revient à faire une division par une puissance de 2)</li> </ul>

### 2.3.3 Les multiplications

Une multiplication d'un nombre de 32 bits par un nombre de 32 bits donne potentiellement un nombre de 64 bits.

$$32b * 32b \rightarrow 32b$$

Pour les petites valeurs des opérandes (quand le résultat tient dans 32 bits, ou quand les 32 bits de poids fort ne sont pas importants), il existe l'instruction `MUL`.

$$32b * 32b \rightarrow 64b$$

Pour les multiplications  $32b * 32b \rightarrow 64b$  il existe 2 types d'instructions :

- `SMULL` Signed Multiply Long
- `UMULL` Unsigned Multiply Long

Pour ces instructions, le résultat se trouve dans 2 registres de 32 bits.

Il existe des instructions effectuant automatiquement l'addition du résultat de la multiplication avec un ou plusieurs autres registres (32 bits ou 64 bits). Ces instructions sont particulièrement utiles en traitement du signal. Les instructions correspondantes sont `MLA`, `SMLAL`, `UMLAL`, ....

Le compilateur ou les bibliothèques utilisées vont directement utiliser les instructions les plus performantes pour accomplir une tâche. Quand l'instruction n'est pas disponible sur un microcontrôleur, le compilateur génère une suite d'instructions élémentaires.

## 2.4 Traitement des instructions

La figure 2.3 montre le fonctionnement (simplifié) de la gestion des instructions par un microcontrôleur.

- (A) Chargement d'une instruction depuis la mémoire



- (B) Chargement éventuel d'un ou des opérandes depuis la mémoire
- (C) Calcul
- (D) Stockage en mémoire

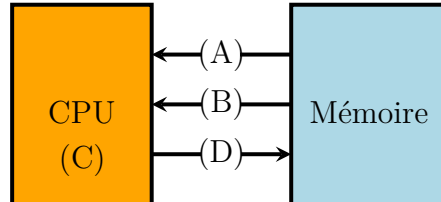


FIGURE 2.3 – Traitement des instructions

### 2.4.1 Type d'architecture

On distingue historiquement deux types d'architectures. Mais avec les microcontrôleurs modernes, on peut avoir des solutions hybrides

**Harvard** L'architecture Harvard sépare physiquement les bus d'adresse et de données pour les instructions et pour les données. Il est ainsi possible d'accéder simultanément aux instructions et aux données. Les instructions peuvent être dans une mémoire en lecture-seule. Cette architecture permet de plus grandes performances pour une fréquence fixée. Elle est utilisée pour beaucoup de microcontrôleurs dans une version modifiée.

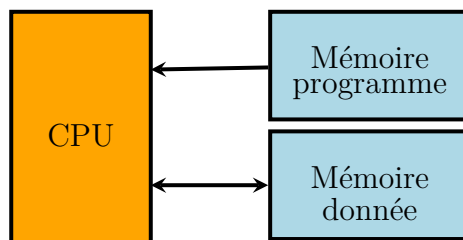


FIGURE 2.4 – Architecture Harvard

**Von Neumann** L'architecture Von Neumann<sup>8</sup> se caractérise par le fait que le bus d'adresse et de données est partagé par les instructions et les données. Un programme peut se modifier lui-même car ses instructions sont stockées dans une zone mémoire standard.

---

8. John von Neumann, 1903-1957

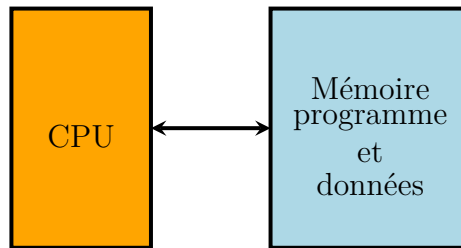


FIGURE 2.5 – Architecture Von Neumann

Avec le **STM32F401**, nous sommes en présence d’une architecture Harvard, les figures 1.11 et 1.12 montrent les deux bus (instructions et data).

### 2.4.2 Organisation des instructions

Le microcontrôleur ne comprend qu’un nombre limité d’instructions.

Elles codent une action élémentaire à effectuer (addition, multiplication, décalage, opération et/ou/non, ...) et le ou les opérandes sur lesquels vont s’appliquer l’action. La taille des instructions varie en fonction des architectures (quelques octets) et des types de microcontrôleur.

Certains registres (des zones mémoires spéciales du microcontrôleur) stockent des informations nécessaires à l’exécution des instructions :

**PC** : le **Program Counter**, il contient l’adresse de l’instruction courante

**SP** : le **Stack Pointer**, le pointeur de pile, permettant de stocker/retrouver de l’information

**Status** : différents registres contiennent l’état en cours, retenue, zéro, ...

Le listing 2.3 montre un petit programme en C (ce programme ne fait rien de spécial).

Ce programme a été compilé pour un ARM avec des options permettant de récupérer le code assembleur généré et de le mixer avec le code original en C. Cela donne le listing 2.4. Ce code a été compilé sans options d’optimisation, sinon la majorité des lignes auraient été supprimées.

Sur le listing 2.4 on retrouve les opcodes des instructions, par exemple l’instruction :

`push r7` a un opcode de `000080B4`, codant à la fois l’instruction, les paramètres sur lesquels elle s’applique et des informations conditionnelles. La liste des mnémoniques des instructions se trouvent dans le document [\[10\]](#).

```
int main(void)
{
    int i = 3;
    int j;
    j = i+1;
}
```

Listing 2.3 – Exemple de programme en C

```

.LFB0:
.file 1 "../src/main.c"
:../src/main.c **** /*
:../src/main.c **** * main.c
:../src/main.c **** *
:../src/main.c **** * Created on: 12 sept.
:../src/main.c **** * Author: ppardo
:../src/main.c **** */
:../src/main.c ****
:../src/main.c ****
:../src/main.c **** int main(void)
:../src/main.c **** {
    .loc 1 10 0
    .cfi_startproc
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
0000 80B4    push {r7}
    .cfi_def_cfa_offset 4
    .cfi_offset 7, -4
0002 83B0    sub sp, sp, #12
    .cfi_def_cfa_offset 16
0004 00AF    add r7, sp, #0
    .cfi_def_cfa_register 7
:../src/main.c **** int i = 3;
    .loc 1 11 0
0006 0323    movs r3, #3
0008 7B60    str r3, [r7, #4]
:../src/main.c **** int j;
:../src/main.c **** j = i+1;
    .loc 1 13 0
000a 7B68    ldr r3, [r7, #4]
000c 0133    adds r3, r3, #1
000e 3B60    str r3, [r7]
0010 0023    movs r3, #0
:../src/main.c **** }
    .loc 1 14 0
0012 1846    mov r0, r3
0014 0C37    adds r7, r7, #12
    .cfi_def_cfa_offset 4
0016 BD46    mov sp, r7
    .cfi_def_cfa_register 13
    @ sp needed
0018 5DF8047B ldr r7, [sp], #4
    .cfi_restore 7
    .cfi_def_cfa_offset 0
001c 7047    bx lr
    .cfi_endproc
.LFE0:
001e 00BF    .text
.Letext0:

```

Listing 2.4 – Exemple de programme assemblé (extrait)

La programmation en langage assembleur reste un exercice complexe. Et il est difficile de se mesurer à la capacité d'un bon compilateur. Il est toujours possible d'analyser le code assembleur généré, mais il sera difficile d'en écrire et surtout de le maintenir.

La section suivante 3.1 présente des outils modernes de développements, ils sont destinés à laisser le développeur se concentrer sur les fonctionnalités à remplir par son application, et à s'affranchir des tâches qui sont mieux réalisées par des outils informatiques.



# 3

## Développement

*Ce chapitre traite des outils utilisés pour le développement de logiciels à destination du **STM32F401**.*

*“If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.”*

- Edsger Dijkstra,

### 3.1 Notions d’IDE

Les IDE (Integrated Development Environment) sont l’environnement de travail du développeur d’application.

#### 3.1.1 Pourquoi un IDE ?

L’IDE va concentrer en un seul environnement tous les outils nécessaires :

- Un éditeur de code source (avec coloration syntaxique, code-completion, warning intelligents en fonction du langage, ...)
- Une interface vers les outils de compilation, avec une gestion des paramètres et des options.
- Un lien avec le débogueur, pour suivre l’exécution du programme en pas à pas, avec un affichage des variables, une gestion des registres et des breakpoints.

Mais on peut trouver aussi des fonctions plus avancées :

- Un gestionnaire de versions de code (intégré ou externe)
- Un gestionnaire de fichier (pour copier, renommer, supprimer, ...)
- Une représentation graphique des liens entre les modules ou les classes
- ...

On peut trouver des IDE dédiés à des langages de programmation, ou à des cibles particulières.

Beaucoup d'IDE sont basés sur **Eclipse**, qui est un IDE reconfigurable et qui, par un mécanisme de plugin, permet d'être adapté à un langage de programmation, mais permet aussi de choisir le type de compilateur et de debugger utilisés.

### 3.1.2 Alternative à l'utilisation d'un IDE

L'utilisation d'un IDE n'est pas obligatoire, mais c'est un bon outil pour toutes les tâches que doit effectuer un développeur. Les alternatives sont l'utilisation d'outils de base :

- Un éditeur de texte (même **Notepad**)
- Un compilateur en ligne de commande (**GCC**)
- Un débogueur en ligne de commande (**GDB**)

### 3.1.3 Un IDE fait maison

À partir de simples outils logiciels, il est possible de se construire un IDE ayant toutes les caractéristiques d'un environnement dédié.

Il est ainsi possible d'intégrer les fonctions de compilation et de débogage à des éditeurs de textes standards (**vim**, **Emacs**, **Notepad++**, **SublimeText**, **Visual Studio Code**, ...).

Le fait d'avoir ce type d'environnement peut être un gage de pérennité pour les projets, en maîtrisant complètement la chaîne de compilation/débogage.

### 3.1.4 Choix de l'IDE

Le choix de l'environnement peut être contraint par la présence d'outils dédiés à un environnement matériel. Les habitudes du développeur ou du client peuvent aussi jouer, l'apprentissage d'un environnement peut demander un certains temps (apprentissage des raccourcis claviers, maîtrise des options, ...).

Les IDE dédiés sont souvent basés sur une plateforme **Eclipse**, qui encapsule les appels à des outils dédiés à la cible utilisée. C'est ce que nous allons utiliser ici avec **STM32CubeIDE**.



## 3.2 Utilisation de STM32CubeIDE

### 3.2.1 Téléchargement

L'application STM32CubeIDE se télécharge directement depuis le site de ST. L'application est disponible pour Linux et Windows.

### 3.2.2 Contenu de l'application

STM32CubeIDE est un environnement de développement basé sur Eclipse. Mais il contient aussi des Wizards (assistants) pour générer automatiquement du code en fonction des paramètres choisis dans un environnement graphique.

Les kits de développement ST sont connus (ou téléchargés) par l'application. Les périphériques existants dans le kit sélectionné seront automatiquement ajouté dans l'environnement et pourront être configuré très simplement par l'intermédiaire de menus.

### 3.2.3 Développer pour un microcontrôleur

Avec STM32CubeIDE nous allons développer du code pour un STM32 :

- Cross compilation, la cible est un ARM (bien que les outils tournent sous Windows ou Linux)
- Environnement d'exécution simple
- Exécution pas à pas simplifiée (GDB)
- Affichage des sorties sur liaison série
- Intégration avec les bibliothèques STM32
- Environnement Eclipse, stable, efficace, et connu

### 3.2.4 Debug sur liaison série

Dans un système embarqué il n'y a en général pas d'écran ou de moyen de communication simple pour tracer des informations (des logs de fonctionnement par exemple). On utilisera souvent, lors des phases de tests, une simple liaison série pour sortir de l'information (une trace d'exécution).

Ce mécanisme est relativement simple à mettre en œuvre :

- La carte Nucléo connectée à un PC est reconnue (entre autre) comme un port série virtuel.
- Nous pouvons facilement configurer des GPIO de la carte Nucléo pour avoir une fonction alternative USART et agir en tant que liaison série.
- Il est possible de rediriger les `printf`<sup>1</sup> vers une liaison série.

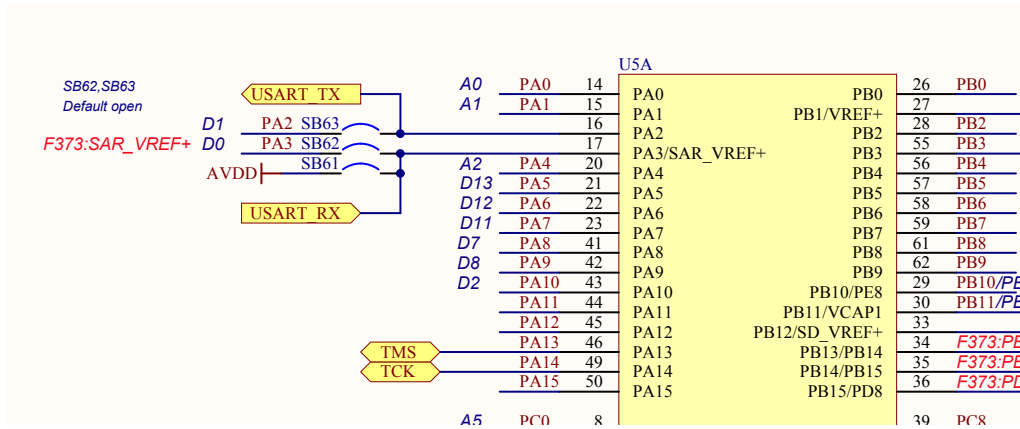


FIGURE 3.1 – GPIO et USART

La configuration complète sera décrite dans les TD, dans les exercices liés à la liaison série.

### 3.3 Bibliothèques CMSIS, StdPeriph, HAL, LL, ...

Il y a plusieurs possibilités pour accéder aux registres du microcontrôleur. L'équilibre étant à trouver entre la simplicité d'écriture, les performances, la portabilité, la présence de bugs (et la confiance dans les bibliothèques utilisées) ....

#### 3.3.1 Accès direct

L'accès direct aux registres est la solution la plus simple. Cela nécessite de lire la documentation du composant, et peut engendrer un temps d'acclimatation à chaque changement de composants (la documentation des registres du STM32 fait environ 840 pages).

Cela reste quand même la meilleure solution pour :

- Comprendre comment fonctionne exactement le microcontrôleur
- Avoir des performances optimales (il n'y a pas d'appel de fonction)
- Avoir une documentation exhaustive (mais avec peu d'exemples)

Une fonction à réaliser va nécessiter de programmer plusieurs registres. Un registre peut contenir plusieurs fonctions, il faudra masquer les valeurs lors des écritures.

1. En surchargeant la fonction `int __io_putchar(int ch)`

La portabilité ne sera pas assurée d'un microcontrôleur à l'autre (même de famille similaire).

### 3.3.2 CMSIS

CMSIS est l'acronyme de **Cortex Microcontroller Software Interface Standard**.

C'est une interface standard pour les fonctions communes aux microcontrôleurs à cœur ARM. Par contre, l'accès aux périphériques (SPI, USART, ...) devra ce faire par les registres.

Pour une même gamme de microcontrôleur, l'abstraction offerte par CMSIS peut être une aide au portage.

CMSIS ne fournit normalement pas d'implémentation, mais plutôt une interface commune vers des fonctionnalités sous-jacentes (OS temps réel par exemple).

L'avantage d'utiliser CMSIS est la portabilité des fonctionnalités de haut-niveau.

### 3.3.3 SPL

La SPL est l'acronyme de **Standard Peripheral Library** (on trouve aussi **StdPeriph**). C'est une bibliothèque de fonctions qui encapsulent l'accès aux registres des périphériques. C'est l'équivalent du CMSIS, mais dédié aux périphériques, et uniquement pour les microcontrôleurs STM.

La SPL est amenée à disparaître car elle n'est plus suivie par ST qui a tendance à promouvoir la HAL (et la **Low Level Library**). Il n'y aura pas de SPL pour les nouveaux microcontrôleurs.

Les deux listings suivants (3.2 et 3.1) sont similaires :

- Ils autorisent l'horloge pour le port GPIOA
- Ils configurent PA5 comme une sortie. Les paramètres sont identiques.

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

GPIO_InitTypeDef g;
g.GPIO_Pin = GPIO_Pin_5;
g.GPIO_Mode = GPIO_Mode_OUT;
g.GPIO_OType = GPIO_OType_PP;
g.GPIO_Speed = GPIO_Speed_100MHz;
g.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOA, &g);
```

Listing 3.1 – Configuration avec StdPeriph

```
RCC->AHB1ENR |= 1;

GPIOA->MODER &= 0xFFFFF3FF;
GPIOA->MODER |= 0x400;
GPIOA->OTYPER &= 0xFFFFFDF;
GPIOA->OSPEEDR |= 0xC00;
GPIOA->PUPDR &= 0xFFFFF3FF;
GPIOA->ODR &= 0xFFFFFDF;
```

Listing 3.2 – Configuration sans `StdPeriph`

Avec les environnements modernes, qui proposent de la coloration syntaxique, de l’auto-complétion, ... l’utilisation d’une bibliothèque comme `StdPeriph` permet de gagner beaucoup de temps.

### 3.3.4 Hardware Abstraction Layer

La librairie HAL est utilisée par le générateur de code automatique STM32CubeIDE. Il permet graphiquement de configurer son microcontrôleur, d’avoir des embryons de fonctions et une structure de code pour la gestion des périphériques.

Les principaux inconvénients de la HAL sont :

- Des bugs (les fonctions ne sont pas exemptes de problèmes)
- La surcouche peut entraîner des soucis de performances.
- La courbe d’apprentissage du HAL. Les fonctions ne sont pas toujours logiques pour un développeur qui connaît les registres.
- Le développeur n’a pas un contrôle direct sur son microcontrôleur. Un appel de fonction HAL peut entraîner des modifications dans plusieurs registres. Les effets de bord ne sont pas tous toujours recommandés.

L’exemple 3.3 reprend le même type de code que précédemment, mais utilise la HAL pour configurer une GPIO.

```
__HAL_RCC_GPIOA_CLK_ENABLE();
GPIO_InitTypeDef GPIO_InitStruct = {0};
GPIO_InitStruct.Pin = GPIO_Pin_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Listing 3.3 – Configuration avec la HAL

### 3.3.5 Low Level Library

La LLL (ou LL) est plus dans l'esprit de la SPL. Elle encapsule les registres, mais reste petite et optimisée. Elle utilise des macros pour ne pas perdre en performances. La LLL est moins portable que la HAL.

### 3.3.6 Choix d'une librairie

Le choix est difficile, et va dépendre du type de projet :

- Si le microcontrôleur ne doit pas évoluer (et si la LLL est disponible), l'utilisation de la LLL simplifie l'utilisation des registres des périphériques et permet rapidement de faire des tests, du prototypage, et éventuellement de regarder quels sont les registres sous-jacents utilisés.
- L'utilisation de HAL permet d'encapsuler l'accès aux registres, mais permet aussi éventuellement un portage plus facile vers un autre microcontrôleur, ceci au dépend d'une complète maîtrise du code, et dans certains cas d'une perte de performance.

Pour les librairies, toutes les fonctions ne sont pas implémentées, et il peut être nécessaire de passer du temps à analyser la librairie pour comprendre son fonctionnement. C'est toujours un bon exercice de regarder comment est fait le code existant pour comprendre la philosophie et les choix effectués.

Mon choix personnel pour des projets d'envergure est d'utiliser en partie CMSIS, et de développer mes propres pilotes pour les périphériques. Cela permet d'avoir un contrôle total sur le code généré. L'effort initial peut être potentiellement plus important par rapport à l'utilisation de librairies existantes, mais à terme, la confiance dans le code développé permet de compenser ce temps d'apprentissage. Les pilotes de périphériques, si ils sont bien développés, peuvent ensuite être réutilisés de projets en projets.

Quand il s'agit de prototypage ou de tests, l'utilisation de librairies existantes permet de réduire les délais.

Un bon compromis peut passer par l'utilisation de librairies, tout en effectuant une relecture de code pour anticiper les soucis potentiels, détecter les points d'optimisation. Et de n'écrire du code spécifique que si cela est vraiment nécessaire.

## 3.4 Utilisation du programmeur autonome de STM32 : STM32 ST-LINK Utility

Ce logiciel se télécharge directement sur le site <http://www.st.com/en/embedded-software/stsw-link004.html>.

### 3.4.1 Installation

Le fichier téléchargé est un zip, une fois extrait vous pouvez lancer l'exécution du fichier STM32 ST-LINK Utility v4.0.0 setup.exe.

L'installation suit la procédure standard des installations Windows.

### 3.4.2 Utilisation

Après un double-click sur l'icône vous devez voir apparaître l'interface de l'application (voir figure 3.2).

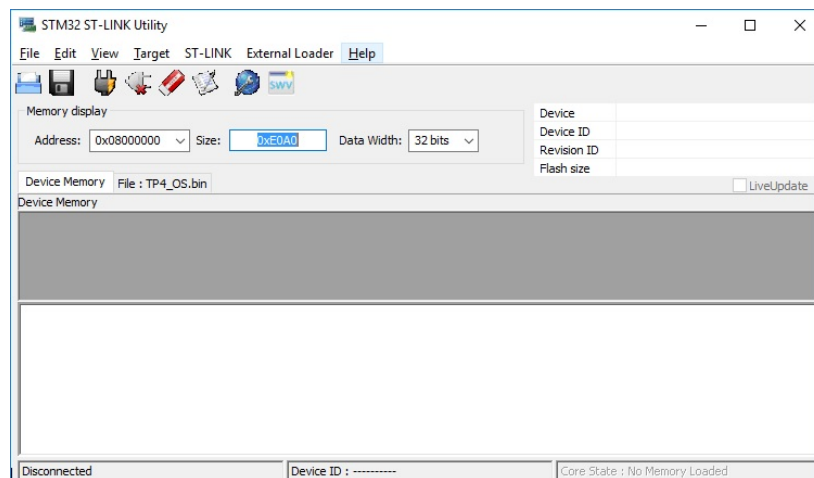


FIGURE 3.2 – Interface STM32 ST-LINK Utility

#### 3.4.2.1 Vérification de la connexion

Le menu Target/Connect permet de vérifier que la carte STM32 est bien reconnue par le PC.

#### 3.4.2.2 Chargement d'un fichier

Il faut indiquer quel est le fichier à charger, le menu à utiliser est **File/Open file**, qui permet de sélectionner un fichier généré par la compilation (un fichier **.bin**).

#### 3.4.2.3 Programmation

Le menu **Target/Program & Verify** permet d'envoyer le code préalablement sélectionné vers la cible.

### 3.5 Cycle développement / compilation / programmation

Le cycle habituel est :

1. Développement du logiciel par ajout de fichiers, fonctions, et codes, ...
2. Compilation des codes sources (génération d'un fichier **.bin**)
3. Envoi de ce code vers la cible

Pour programmer la flash du microcontrôleur, il y a plusieurs possibilités, qui correspondent à des moments du cycle de vie (ou de la maturité) du développement :

- Avec l'environnement de développement
  - Menu **Debug As ...**
  - Menu **Run As ...**
- Avec un outil externe
  - **STM32 ST-LINK Utility**
- Grâce à la partie sécable de votre carte de développement
  - Avec le USB device **NodeF401\_RE**
- Grâce à un moniteur 'maison' ou fourni par le microcontrôleur utilisé
  - A vous de faire un logiciel qui puisse se reprogrammer

Les cas d'utilisation de ces méthodes sont par exemple :

- L'environnement de développement permet de faire de la programmation, du débogage, du suivi en pas à pas pendant les premiers temps de la vie du programme.
- L'outil externe **STM32 ST-LINK Utility** permet :
  - D'avoir un fichier commun programmé dans plusieurs cartes complètement identiques.
  - De s'affranchir des outils de développement quand on est en 'production'.

- Le USB device n'est utilisable que sur la carte **Nucleo** et nécessite du hardware supplémentaire, mais peut être une bonne architecture pour une carte de démonstration ou un prototype.
- Le moniteur 'maison' doit être utilisé quand le firmware doit pouvoir être mis à jour durant la vie du produit. Certains microcontrôleurs contiennent nativement un moniteur.

## 3.6 Méthodes agiles

Les méthodes agiles de développement logiciel sont parfaitement applicables aux développement d'applications embarquées.

Les bonnes pratiques de développement recommandées par la méthode XP [11] (par exemple) permettent d'augmenter pour le développeur la confiance dans le code généré. En toutes occasions, le développement de tests unitaires sera une aide pour un développement de logiciel fiable et robuste.

Des fonctions simples de tests unitaires peuvent être par exemple :

- Un fichier `unittest.h`

```
#ifndef UNITTEST_H_INCLUDED
#define UNITTEST_H_INCLUDED

extern void initTests(void);
extern void assertTest(int v);
extern int reportTests(void);
extern void assertTestWL(int v, char *pcFile, int iLine);

#define assert(a) assertTestWL((a), (char*)__FILE__, __LINE__)

#endif // UNITTEST_H_INCLUDED
```

Listing 3.4 – Fichier d'en-tête pour des tests unitaires simples

- Un fichier `unittest.c`, suivant la cible, les `printf` peuvent être supprimés. Seule le compte des erreurs sera surveillé. Eventuellement, on peut renvoyer le numéro de la ligne ayant causé l'erreur.



```
#include <stdio.h>
#include <stdlib.h>

#include "unittest.h"

unsigned nbTests;
unsigned nbErrors;

unsigned firstErrorLine;

/** \brief Initialise les variables utilisée par les fonctions de tests
↪ unitaires
 *
 * \param void
 * \return void
 */
void initTests(void)
{
    nbTests = 0;
    nbErrors = 0;
    firstErrorLine = 0;
}
```

Listing 3.5 – Tests unitaires simples, initialisation

```
/** \brief Effectue un test
 * Le compteur de test est incrémenté
 * Le compteur d'erreur est incrémenté éventuellement (si il y a une erreur)
 * \param v int vrai ou faux
 * \return void
 * \sa assertTestWL, assert
 */
void assertTest(int v)
{
    nbTests++;
    if (!v)
    {
        nbErrors++;
    }
}

/** \brief Effectue un test, mais affiche des informations complémentaires
 * Le compteur de test est incrémenté
 * Le compteur d'erreur est incrémenté éventuellement (si il y a une erreur)
 * ↪ et un message de localisation est affiché
 * \param v int vrai ou faux
 * \param pcFile char* le fichier où s'est produit l'erreur
 * \param iLine int la ligne où a eu lieu l'erreur
 * \return void
 * \sa assertTest, assert
 */
void assertTestWL(int v, char *pcFile, int iLine)
{
    nbTests++;
    if (!v)
    {
        nbErrors++;
        if (0 == firstErrorLine)
        {
            firstErrorLine = iLine;
        }
        printf("Erreur[%03u], %s(%d)\n", nbErrors, pcFile, iLine);
    }
}
```

Listing 3.6 – Tests unitaires simples, fonctions de tests

```
/** \brief Affiche un rapport sur les tests effectués
 * En particuliers Ok ou Ko, le nombre de tests, et le nombre d'erreurs (si il
 ↪ y en a)
 * \param void
 * \return int le nombre d'erreurs
 */
int reportTests(void)
{
    if (nbErrors == 0)
    {
        printf("\nOk, %u tests\n", nbTests);
    }
    else
    {
        printf("\nKo, %u tests, %u erreurs\n", nbTests, nbErrors);
    }
    return nbErrors;
}
```

Listing 3.7 – Tests unitaires simples, rapport

- Un exemple de tests unitaires

```
// avant les tests
initTests();

// des tests, tout peut être testé, par une suite de assert(...)
assert((1+2) == 3);
assert(VeryComplexFunction(7, 5) > 123)
assert(0 == strcmp("azerty", pcStringToTest));

// affichage du rapport sur les tests effectués
reportTests();
```

Listing 3.8 – Fichier d'exemple pour des tests unitaires simples



# 4

## Notions de temps réel

*Ce chapitre introduit les systèmes temps réels multitâches.*

*“If 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself “Dijkstra would not have liked this”, well, that would be enough immortality for me.”*

- Edsger Dijkstra,

### 4.1 Pourquoi un système d’exploitation temps réel ?

Une application embarquée peut parfaitement fonctionner sans utiliser de système d’exploitation temps réel. Mais le développement d’une application complexe va être grandement facilité quand on va utiliser les abstractions fournies par un système d’exploitation temps réel. Le développeur (ou l’architecte du système) va pouvoir se concentrer sur la partie fonctionnelle de l’application, en réutilisant des outils fiables déjà intégrés.

Il est courant de redévelopper (ou d’adapter) à chaque nouvelle application, des parties du logiciel pour gérer au mieux :

- La communication entre des parties du programme.
- La bonne gestion des interruptions.
- Des mécanismes de synchronisations et d’attente.
- La séparation du programme en module, permettant les tests et la réutilisabilité.
- ...

Ce que propose un système d’exploitation temps réel, est de fournir des services fiables, efficaces, éprouvés dans les catégories suivantes :

- Séparation du programme en modules (tâches), en laissant le concepteur maître des priorités.
- En fournissant des services de communication, de modules à modules, en garantissant l'intégrité des données transmises et en s'affranchissant des problèmes liés à la concurrence d'accès.
- En gérant au mieux les temps d'attente, plus aucune ressource CPU n'est utilisée pour faire du polling.
- En permettant facilement de passer l'application en basse consommation.
- Une gestion des interruptions déferées, permettant à la fois de ne pas rater d'interruptions, tout en continuant à servir les tâches prioritaires.
- La protection des sections et ressources critiques avec des mécanismes simples.

## 4.2 Quelques définitions

**Embarqué, Enfoui, Embedded** [13] Un système embarqué est défini comme un système électronique et informatique autonome, souvent temps réel, spécialisé dans une tâche bien précise. Le terme désigne aussi bien le matériel informatique que le logiciel utilisé. Ses ressources sont généralement limitées. Cette limitation est généralement d'ordre spatial (encombrement réduit) et énergétique (consommation restreinte).

**Temps réel, Real time** [14] Un système temps réel est une application ou plus généralement un système pour lequel le respect des contraintes temporelles dans l'exécution des traitements est aussi important que le résultat de ces traitements. Ces systèmes sont utilisés pour contrôler des processus physiques en suivant leur évolution au rythme de leur déroulement. On distinguera :

- Les systèmes temps réel durs, pouvant assurer une garantie de temps de réaction.
- Les systèmes temps réel probabilistes. Qui font au mieux !

**Multitâche, multitask** [12] Un système d'exploitation est multitâche s'il permet d'exécuter, de façon apparemment simultanée, plusieurs programmes informatiques. On distinguera :

- Les systèmes multitâches préemptifs
- Les systèmes multitâches non-préemptifs (coopératifs)

## 4.2.1 Une application embarquée temps réel

### 4.2.1.1 Pourquoi embarquée ?

Une application embarquée sera en général dédiée à une application, et sera limitée dans les fonctionnalités qu'elle aura à remplir.

Cela peut être le système audio/gps d'un véhicule, le contrôle d'un appareil ménager, une télécommande, ....

Le système embarqué sera normalement plus petit et moins performant qu'un ordinateur destiné à une application générique. Les usages seront différents, la consommation sera réduite, et le prix sera largement inférieur.

Un microcontrôleur 8 bits, fonctionnant à quelques MHz, avec quelques kB de RAM et de flash sera suffisant pour de simples applications.

Du fait de l'évolutions des capacités de microcontrôleurs, et la demande croissante de plus de puissance embarquée, on peut trouver maintenant des applications sur des microcontrôleurs 32 bits, fonctionnant à plusieurs centaines de MHz, avec 1 GB de RAM et plusieurs GB de flash.

Quelques particularités des systèmes embarqués :

- Il peut n'y avoir aucune interface, ou une interface minimaliste
- Il y a des contraintes de fiabilité du fait des actions sur le monde physique
- Ils sont dédiés à des tâches prédéterminées (ou à évolution lente)
- On peut trouver des modes dégradés, garantissant la sécurité et l'intégrité en cas de détection de pannes (alimentation, composant en erreur, ...)

### 4.2.1.2 Pourquoi temps réel ?

Le temps réel n'est pas synonyme de rapidité.

Le temps réel va être la garantie qu'un évènement se produira au moment prévu, ou que le système réagira dans un temps connu à un stimuli.

Cela peut se comprendre aisément en prenant l'exemple d'un contrôleur d'Airbag, quand le déclenchement doit être effectué, il n'est pas question d'attendre ou d'être ralenti par une autre tâche s'exécutant sur le microcontrôleur. Il faut pouvoir garantir la latence du déclenchement (et en général le séquençage du déclenchement des airbags nécessaires).

Un pacemaker est un autre exemple simple, ou la génération d'impulsions doit être effectuée avec un rythme régulier et maîtrisé.

La vitesse n'intervient pas. L'important est la garantie d'une latence connue entre le moment où un évènement se déclenche, et le moment où il est pris en compte par le système..

### 4.2.2 Système critique

Un système embarqué va en général interagir avec son environnement extérieur. Contrairement à une application standard sur un ordinateur, il devra fonctionner sans intervention humaine et remplir les fonctions prévues au mieux.

Des pannes peuvent survenir, mais tout doit être mis en œuvre pour que les dysfonctionnements n'empêchent pas les fonctions principales du système. Suivant les domaines (médical, avionique, ...), des normes de développement seront imposées, pour le matériel et pour le logiciel. En cas de systèmes critiques, on pourra trouver de la redondance de systèmes, des équipements durcis (compatibilité électromagnétique), ...

### 4.2.3 Encombrement et consommation

Un appareil va devoir effectuer un grand nombre d'opérations.

1. Une approche simple serait de séparer physiquement chaque fonction du système. Cela revient à avoir :
  - Plusieurs microcontrôleurs (un par fonction)
  - Plusieurs alimentations
  - Plusieurs horloges
  - Un moyen de communication entre les microcontrôleurs
  - Un moyen de synchronisation entre les microcontrôleurs
  - Un contrôleur de séquençement de démarrage
  - Plusieurs firmwares à valider et à gérer
2. Une approche plus intégrée serait :
  - Un seul microcontrôleur
  - Une seule alimentation
  - Un partage du temps entre les différentes tâches à accomplir
  - Un moyen de communication entre les tâches (interne)
  - Un moyen de synchronisation entre les tâches (interne)
  - Un seul firmware à valider et à gérer

C'est cette deuxième approche qui est bien sûr retenue pour les raisons suivantes :

- Un encombrement moindre et une réduction de la consommation
- Une maintenance facilitée
- La mutualisation permet de réduire la consommation
- De meilleures performances à fréquence constante (la communication dans un microcontrôleur est plus efficace que par un bus externe)



Mais cela se fait au détriment d'une augmentation de la complexité :

- Le temps est partagé, mais des priorités doivent être définies pour garantir que les fonctions vitales de l'application ne seront pas ralenties ou arrêtées par d'autres fonctions.
- Le partage des périphériques nécessite des arbitrages
- Le développement et la mise au point sont plus complexes

⇒ l'utilisation d'un système d'exploitation temps réel va nous aider à réaliser tout cela.

A titre d'exemple, le **Apollo Guidance Computer** (1960-...) montre bien l'intérêt de la mutualisation et du partage du temps machine pour réduire l'encombrement.

Une idée principale : Avec moins de hardware, il y a plus de place pour d'autres équipements.

C'est :

- Un système multitâches (8!)
- Avec une mémoire ROM de 32000 mots de 16 bits
- Une mémoire de 2000 mots de RAM
- Tout ceci pour environ 35 kg



FIGURE 4.1 – Apollo Guidance Computer

Les codes sources sont disponibles sur [GitHub](#).

Plus proche (temporellement) de nous, on peut voir l'effort d'intégration réalisé dans les quelques appareils de la vie quotidienne suivants :



FIGURE 4.2 – Des appareils modernes

#### 4.2.4 Sécurité

Il y a de plus en plus d'objets connectés (IoT), et les moyens d'attaques sont variés :

- La connexion à un réseau est un vecteur d'attaque supplémentaire.
- L'injection de codes modifiés lors des mises à jour (prévues par le fournisseur, en OTA, ...).
- L'accès physique facilité par la profusion d'appareils connectés (on ne maîtrise pas toujours la protection physique du système). Les systèmes peuvent être physiquement répartis.
- L'inexpérience des utilisateurs (mauvais mots de passe).
- L'inexpérience des développeurs (trop de confiance).
- ...

Les objets connectés sont un vecteur d'attaque (le maillon faible) pour un système entier du fait de l'interconnexion des systèmes.

**Exemple :** Le 21 octobre 2016 l'attaque par un virus depuis des objets connectés a fait *tomber* le site de Dyn (DNS), entraînant des arrêts de service chez Netflix, Spotify, CNN, Airbnb, Amazon, Twitter, Reddit... Les retombées économiques (immédiates, mais aussi la perte de confiance) peuvent être importantes. Les ransomware sont un risque important, et les objets connectés sont un vecteur d'attaque à prendre en compte.

### 4.2.5 Fiabilité

Le but de notre système va être de remplir les tâches prévues :

- En continu
- Pendant longtemps
- Sans interventions humaines
- D'accepter un mode dégradé éventuel
- ...

Un système fiable le sera au niveau matériel et aussi au niveau logiciel.

Au niveau matériel, des règles doivent être mise en place :

- Pas de pièces mobiles, minimisation des équipements
- Redondance
- Tests environnementaux
- Normes
- Bonnes pratiques
- Maintenance prédictive
- ...

Mais le problème est similaire au niveau logiciel :

- Certification des outils
- Méthode de programmation, règles (MISRA, ...)
- Logiciel tolérant pour permettre un mode dégradé en cas de soucis matériel
- Chiffrement de firmware
- Preuve de logiciel
- Expérience des équipes de développement
- ...



# 5

## Temps réel Multitâches avec FreeRTOS

*Ce chapitre traite de l'utilisation de **FreeRTOS** et de la manière dont les concepts des systèmes temps réel ont été implémentés.*

### 5.1 Concepts de base

Les systèmes d'exploitation temps réel multitâches essaient de résoudre les problèmes suivants :

- Multitâche : Réduire le nombre de matériel utilisé (voir 4.2.3), en regroupant les logiciels sur une même plateforme matérielle.
- Garantir un temps de réponse à des stimuli extérieurs, garantir la fréquence des événements périodiques.

Pour le multitâche, malgré le fait que le microcontrôleur soit partagé entre plusieurs tâches, on va définir des priorités :

- Les tâches les plus prioritaires ne doivent pas être gênées (ralenties ou arrêtées) par des tâches moins prioritaires.
- On accepte qu'une tâche de faible priorité soit bloquée par une tâche plus prioritaire.

Pour le temps réels :

- On souhaite garantir que le code associé à un événement extérieur (interruption) soit géré dans un temps connu (la latence).
- On souhaite que les événements périodiques aient le moins de dérive possible (pas de jitter).

Ces contraintes multitâches/temps réels sont contradictoire et devront être traitées au cas par cas. Le système d'exploitation temps réel fournit au développeur les outils pour bâtir un système fiable et performant.

Mais l'utilisation d'un systèmes d'exploitation multitâches temps réel n'est rien sans la rigueur de développement. Ce ne sont que des outils, et si ils sont mal employés, des résultats désastreux peuvent être obtenus :

- Programme buggé, plantage de l'application, mauvaise performance
- Temps réel non garanti
- Fonctionnalités prévues non assurées
- Programme non maintenable
- ...

### 5.1.1 Architecture

Les systèmes d'exploitation temps réels peuvent se trouver sous la forme d'OS standard (comme **QNX**), comme un sur-ensemble à un OS généraliste (par exemple **VxWorks** sur Windows) ou sous forme de code sources à compiler avec l'application (par exemple **FreeRTOS**).

Nous allons utiliser **FreeRTOS**, cela revient donc à ajouter des fichiers **c** et **h** à notre application qui doit utiliser les mécanismes d'un système d'exploitation temps réel.

Mais les concepts que nous allons retrouver ici seront communs à tous les systèmes d'exploitation temps réel. Les seules différences seront les syntaxes d'appels des fonctions, et la présence ou non d'option pour l'ordonnanceur.

Tous les systèmes d'exploitation temps réel ne sont pas égaux. Certains sont dédiés à certaines fonctionnalités, d'autres sont prévus pour une seule architecture matérielle, ... Le choix ne sera pas toujours simple.

Il existe une version certifiée<sup>1</sup> de **FreeRTOS** (nommée **SafeRTOS**). L'API est similaire à celle de **FreeRTOS**, mais les fonctions ont été validées et réécrites dans le but de certifier le produit. Il est possible de commencer un développement avec **FreeRTOS** et de basculer vers **SafeRTOS** par la suite si cela est nécessaire.

**FreeRTOS** semble plus performant que **SafeRTOS**, qui utilise en standard le MPU<sup>2</sup>.

Il existe aussi une version commerciale **OpenRTOS**, basée sur le même code que **FreeRTOS**, mais offrant des pilotes supplémentaires et du support.

### 5.1.2 Licence

La licence d'utilisation de **FreeRTOS** est de type MIT (en réalité licence Expat).

---

1. IEC 61508-3 SIL 3, ISO 26262 ASIL D

2. Memory Protection Unit



### Texte original de la licence

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions :

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Voici quelques points particuliers de cette licence :

- Gratuite.
- Utilisable dans une application commerciale.
- Sans royalties.
- Sans garantie commerciale (mais des licences **OpenRTOS** et **SafeRTOS** existent).
- Il n'est pas nécessaire de diffuser le code source de l'application utilisant **FreeRTOS**.
- Les modifications apportées à **FreeRTOS** n'ont pas à être diffusées.
- Il n'est pas nécessaire d'indiquer que le produit développé utilise **FreeRTOS** (mais le texte de la licence doit être diffusé si le code source **FreeRTOS** est diffusé).

#### 5.1.3 Code source

**FreeRTOS** est fourni sous forme de code source (des fichiers `h` et des fichiers `c`). Ces fichiers seront compilés avec votre application.

### 5.1.4 Organisation des fichiers

FreeRTOS est distribué sous forme d'un fichier exécutable (actuellement FreeRTOSv10.1.1) depuis le site <https://freertos.org/>. Ce fichier exécutable est en fait un fichier compressé qui extrait tous les fichiers et dossiers de FreeRTOS lors de son lancement.

Cette arborescence de fichiers et dossiers contient tous les codes sources de FreeRTOS, mais aussi les exemples d'applications pour tester le fonctionnement, et les portages vers les différentes cibles supportées.

Vous trouvez aussi des exemples d'applications réseaux, qui, bien qu'elles soient destinées à un petit nombre d'équipement, permettent rapidement d'implémenter des services Telnet, HTTP, ....

#### 5.1.4.1 Recommandation FreeRTOS

La documentation de FreeRTOS recommande d'avoir l'arborescence des fichiers dans un dossier séparé, et de venir pointer les dossiers importants (`include`, `sources` et les fichiers spécifiques à la cible) avec votre environnement de développement.

Ceci a l'avantage de pouvoir faire évoluer FreeRTOS pour l'ensemble de vos projets et d'éviter de dupliquer du code.

Mais ceci peut être un inconvénient pour :

- Archiver un projet (il faut penser à archiver la version courante de FreeRTOS).
- Faire des modifications spécifiques à un projet (elles vont s'appliquer dans tous vos projets).

#### 5.1.4.2 Mes recommandations

Je propose de copier directement les fichiers nécessaires de FreeRTOS pour votre cible dans votre dossier de projet (dans un sous-dossier `freertos` par exemple).

De cette manière :

- Les archivages (`git`) de votre projet contiennent tous les fichiers nécessaires et FreeRTOS est versionné avec le projet.
- Les modifications sont locales et n'impactent pas les autres projets.
- La taille des quelques fichiers est peu importante comparée aux capacités des disques actuels.

Par contre, lors d'une évolution de FreeRTOS, si cela est nécessaire pour plusieurs projets, vous devez tous les modifier.



Pour des raisons de licence, et si vous indiquez que vous utilisez **FreeRTOS**, il est bon de fournir une version complète des codes sources (et pas seulement ceux qui sont inclus dans votre projet).

## 5.1.5 Fichiers de FreeRTOS

### 5.1.5.1 Ajout manuel des fichiers dans un projet

FreeRTOS peut fonctionner avec uniquement les fichiers `C`, `tasks.c`, `list.c` et `queue.c`. Mais pour un fonctionnement normal avec toutes les fonctions disponibles, il faut ajouter dans votre projet les fichiers `C` provenant du dossier `source` et du dossier `include` de FreeRTOS :

```
freertos
├── "tasks.c, tasks.h"
├── "list.c, list.h"
├── "queue.c, queue.h"
├── "timers.c, timers.h"
├── "event_groups.c, event_groups.h"
├── "croutine.c, croutine.h"
├── "stream_buffer.c, stream_buffer.h, message_buffer.h"
├── "FreeRTOS.h"
├── "semphr.h"
├── "portable.h"
└── "stack_macros.h" ..... Qui remplace StackMacros.h devenu obsolète
```

Il faut ensuite ajouter les fichiers destinés au compilateur.

Ces fichiers se trouvent dans le dossier `Source\portable` de FreeRTOS.

Pour le STM32F401, avec un compilateur GCC, les fichiers à récupérer se trouvent dans le dossier `Source\portable\GCC\ARM_CM4F`.

```
freertos
├── "port.c"
└── "portmacro.h"
```

Il faut ensuite récupérer les fichiers du gestionnaire de mémoire choisi. Ils se trouvent dans le dossier `Source\portable\MemMang`. Il n'est pas nécessaire de copier tous les gestionnaires de mémoire, mais si vous souhaitez les avoir tous disponibles, il suffit de les copier et de renommer ceux qui ne sont pas utilisés (avec une extension `.dontuse_c` par exemple).

Les fichiers de gestion mémoire `heap_1.c`, `heap_2.c`, ... correspondent à ceux qui sont décrits au chapitre 6.2.

```
freertos
└─ "heap_1.c"
```

Votre système est maintenant presque complètement configuré. Vous devez maintenant créer un fichier de configuration `FreeRTOSConfig.h`. Le plus simple étant de prendre ce fichier depuis un exemple fourni avec FreeRTOS.

La dernière étape consiste à indiquer à votre environnement de développement le chemin des fichiers de FreeRTOS.

Il n'y a plus qu'à coder...

### 5.1.5.2 Depuis un projet existant

Il est possible d'importer un projet contenant FreeRTOS, c'est ce que nous ferons lors des manipulations.

### 5.1.5.3 Depuis une démo

Il y a un grand nombre de fichiers de démonstration (officiels) disponibles avec la distribution de FreeRTOS. Il y a des exemples utilisant différents compilateurs, microcontrôleurs, ...

Vous pouvez aussi trouver des exemples avancés, utilisant des piles TCP/IP, UDP et des protocoles Telnet, Http ....

Des informations supplémentaires se trouvent dans le manuel de référence [6].

## 5.1.6 Conventions de nommage

### 5.1.6.1 Types de données

Les portages de FreeRTOS sur l'architecture cible définissent 2 types spéciaux `TickType_t` et `BaseType_t`.

**TickType\_t** Ce type sert à stocker les événements temporels. Il peut être de 16 ou 32 bits suivant l'architecture.

**BaseType\_t** Type le plus performant pour l'architecture utilisée. Il sert en général à stocker des booléens, ou de petits entiers (8 bits). Sur une architecture 32 bits, un `uint32_t` sera utilisé.

Les types de bases (`int` par exemple) ne sont normalement pas utilisés. On préférera les types ayant une taille connue (et non dépendante de l'implémentation).

#### 5.1.6.2 Nom des variables

Les variables sont préfixées avec leur type :

**c** Pour les `char`

**s** Pour les `short`

**l** Pour les `long`

**x** Pour les `BaseType_t` et les types complexes (structures, handles, ...)

Cela peut se cumuler avec les préfixes suivants :

**u** Pour les `unsigned`

**p** Pour les `pointeurs`

Une variable dont le nom commence par `uc` sera une variable de type caractère non signé.

#### 5.1.6.3 Nom des fonctions

Les noms des fonctions sont préfixés avec le type qu'elle retourne et avec le fichier dans lequel elles sont définies.

Par exemple :

**vTaskPrioritySet()** Retourne un type `void` et provient du fichier `task.c`

**xQueueReceive()** Retourne un type `BaseType_t` et provient du fichier `queue.c`

**vSemaphoreCreateBinary()** Retourne un type `void` et provient du fichier `semphrtask.c`

Les fonctions privées sont préfixées par `prv`.

#### 5.1.6.4 Nom des macros

Les noms des macros sont en majuscule et préfixés (en minuscule) par le nom du fichier :

Par exemple :

**portMAX\_DELAY** Provient du fichier `portable.h` ou `portmacro.h`

**taskENTER\_CRITICAL** Provient du fichier `task.h`

**pdTRUE** Provient du fichier `projdefs.h`

**configUSE\_PREEMPTION** Provient du fichier `FreeRTOSConfig.h`

**errQUEUE\_FULL** Provient du fichier `projdefs.h` (exception à la règle des noms de fichiers, `projdefs.h` contient les définitions générales)

Les macros courantes suivantes sont aussi définies :

**pdTRUE** 1

**pdFALSE** 0

**pdPASS** 1

**pdFAIL** 0

## 5.2 Systick et FreeRTOS

La base du fonctionnement d'un système d'exploitation temps réel est l'ordonnanceur (ou scheduler) qui va décider quelle est la tâche qui doit devenir active. Le rythme de l'ordonnanceur (le tick système) peut être généré par un timer spécial du microcontrôleur (le Systick sur un STM32, voir en 1.6.7.1), ou par un timer préalablement configuré.



### Modification du fichier `stm32f4xx.h`

Il y a une légère erreur de configuration dans le fichier `stm32f4xx.h`. Il faut modifier la ligne suivante (25 MHz => 8 MHz). Cela se trouve environ en ligne 144, mais a été déjà modifié si vous importez un projet.

```
// #define HSE_VALUE    ((uint32_t)25000000) /*!< Value of the External
↪ oscillator in Hz */
// MODIFICATION CAR sur les nucleo, le external oscillator est à 8MHz
#define HSE_VALUE    ((uint32_t)8000000) /*!< Value of the External
↪ oscillator in Hz */
```

Listing 5.1 – Adaptation à la fréquence de l'oscillateur externe

La période habituelle du Systick est de 1 ms. Mais il peut être configuré à la période désirée. Cela ne change pas le fonctionnement de FreeRTOS (qui va tenir compte de la période).

FreeRTOS utilise les définitions suivantes depuis le fichier `FreeRTOSConfig.h` :

- **configCPU\_CLOCK\_HZ** comme fréquence du microcontrôleur..
- **configTICK\_RATE\_HZ** comme période pour les ticks systèmes de l'ordonnanceur.

En 1.6.4.2 nous avons vu que la fréquence dans nos exemples était de 26 880 000 Hz. Le fichier `FreeRTOSConfig.h` devra contenir des lignes comme dans le listing 5.2 pour avoir des ticks systèmes tous les 1 ms.

```
#define configCPU_CLOCK_HZ      ( 26880000 )
#define configTICK_RATE_HZ     ( ( TickType_t ) 1000 )
```

Listing 5.2 – Configuration de `FreeRTOS`

Le fait de changer le `configTICK_RATE_HZ` change la granularité des temps d’attentes et éventuellement la bascule d’une tâche à une tâche plus prioritaire. Les mécanismes de `FreeRTOS` pour la gestion des interruptions font qu’il est possible de garantir une latence minimale entre une interruption et l’exécution de code utile, pour toutes les valeurs de la périodes du systick (voir en 8.1.8).

Pour cela, à intervalle de temps régulier (dépendant du `configTICK_RATE_HZ`), l’ordonnanceur va :

- Interrompre la tâche en cours d’exécution
- Parcourir la liste des tâches pouvant s’exécuter
- Sélectionner celle qui a la plus haute priorité
- Reprendre l’exécution avec cette nouvelle tâche

La figure 5.1 montre les différents états d’une tâche.

## 5.3 Tâche et descripteur de tâche

Chaque tâche créée dans `FreeRTOS` (plus la tâche `IDLE` qui est créée automatiquement) est définie par un descripteur `TCB_t` (ou `tskTCB` dans les versions plus anciennes).

Ce descripteur est défini comme dans la structure du listing 5.3.

On peut remarquer que des parties de la structure ne sont implémentées que dans certaines conditions, et sont éliminées si des paramètres de configuration ne sont pas utilisés.

Par exemple, si le `MPU` (Memory Protection Unit) n’est pas nécessaire dans l’application, le code correspondant ne sera pas compilé. Cela est un des avantages des systèmes d’exploitations distribués sous forme de codes source : les parties de code non nécessaires ne sont pas compilées et l’empreinte mémoire sera adaptée au plus juste des fonctionnalités.

Le champ `uxPriority`, contient la priorité courante de la tâche. Quand les `Mutex` sont utilisés il existe un autre champ contenant la priorité d’origine

de la tâche. Ceci sera utilisé si une élévation de priorité est nécessaire (ce mécanisme sera précisé en 7.1.2.4).

La structure ne contient pas de champ pour indiquer l'état courant de la tâche (**Running**, **Ready**, **Blocked**, **Suspended**), FreeRTOS gère des listes de tâches dans chaque état. Un changement d'état d'une tâche correspond à une affectation dans une liste différente. Cette architecture simplifie, lors du changement de contexte, la recherche de la prochaine tâche à activer.

Sauf pour l'apprentissage ou les tests, il n'y aura jamais besoin d'effectuer des opérations directement dans cette structure, les fonctions de l'ordonnanceur se chargent d'encapsuler les fonctions nécessaires.

## 5.3 Tâche et descripteur de tâche

```
/*
 * Task control block. A task control block (TCB) is allocated for each task,
 * and stores task state information, including a pointer to the task's context
 * (the task's run time environment, including register values)
 */
typedef struct TaskControlBlock_t
{
    volatile StackType_t *pxTopOfStack; /*< Points to the location of the last item
    ↪ placed on the tasks stack. THIS MUST BE THE FIRST MEMBER OF THE TCB STRUCT. */

    #if ( portUSING_MPU_WRAPPERS == 1 )
        xMPU_SETTINGS xMPUSettings; /*< The MPU settings are defined as part of the port
        ↪ layer. THIS MUST BE THE SECOND MEMBER OF THE TCB STRUCT. */
    #endif

    ListItem_t xStateListItem; /*< The list that the state list item of a task is
    ↪ reference from denotes the state of that task (Ready, Blocked, Suspended ). */
    ListItem_t xEventListItem; /*< Used to reference a task from an event list. */
    UBaseType_t uxPriority; /*< The priority of the task. 0 is the lowest
    ↪ priority. */
    StackType_t *pxStack; /*< Points to the start of the stack. */
    char pcTaskName[ configMAX_TASK_NAME_LEN ]; /*< Descriptive name given to the
    ↪ task when created. Facilitates debugging only. */ /*lint !e971 Unqualified char
    ↪ types are allowed for strings and single characters only. */

    #if ( ( portSTACK_GROWTH > 0 ) || ( configRECORD_STACK_HIGH_ADDRESS == 1 ) )
        StackType_t *pxEndOfStack; /*< Points to the highest valid address for the
        ↪ stack. */
    #endif

    #if ( portCRITICAL_NESTING_IN_TCB == 1 )
        UBaseType_t uxCriticalNesting; /*< Holds the critical section nesting depth for
        ↪ ports that do not maintain their own count in the port layer. */
    #endif

    #if ( configUSE_TRACE_FACILITY == 1 )
        UBaseType_t uxTCBNumber; /*< Stores a number that increments each time a TCB is
        ↪ created. It allows debuggers to determine when a task has been deleted and then
        ↪ recreated. */
        UBaseType_t uxTaskNumber; /*< Stores a number specifically for use by third
        ↪ party trace code. */
    #endif

    #if ( configUSE_MUTEXES == 1 )
        UBaseType_t uxBasePriority; /*< The priority last assigned to the task - used
        ↪ by the priority inheritance mechanism. */
        UBaseType_t uxMutexesHeld;
    #endif
}
```

```

    #if ( configUSE_APPLICATION_TASK_TAG == 1 )
        TaskHookFunction_t pxTaskTag;
    #endif

    #if( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 )
        void *pvThreadLocalStoragePointers[ configNUM_THREAD_LOCAL_STORAGE_POINTERS ];
    #endif

    #if( configGENERATE_RUN_TIME_STATS == 1 )
        uint32_t ulRunTimeCounter; /*< Stores the amount of time the task has spent in
        ↳ the Running state. */
    #endif

    #if ( configUSE_NEWLIB_REENTRANT == 1 )
        /* Allocate a Newlib reent structure that is specific to this task.
        Note Newlib support has been included by popular demand, but is not
        used by the FreeRTOS maintainers themselves. FreeRTOS is not
        responsible for resulting newlib operation. User must be familiar with
        newlib and must provide system-wide implementations of the necessary
        stubs. Be warned that (at the time of writing) the current newlib design
        implements a system-wide malloc() that must be provided with locks. */
        struct _reent xNewLib_reent;
    #endif

    #if( configUSE_TASK_NOTIFICATIONS == 1 )
        volatile uint32_t ulNotifiedValue;
        volatile uint8_t ucNotifyState;
    #endif

    /* See the comments above the definition of
    tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE. */
    #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 ) /*lint !e731 !e9029 Macro has
    ↳ been consolidated for readability reasons. */
        uint8_t ucStaticallyAllocated; /*< Set to pdTRUE if the task is a statically
        ↳ allocated to ensure no attempt is made to free the memory. */
    #endif

    #if( INCLUDE_xTaskAbortDelay == 1 )
        uint8_t ucDelayAborted;
    #endif

    #if( configUSE_POSIX_ERRNO == 1 )
        int iTaskErrno;
    #endif
} tskTCB;

/* The old tskTCB name is maintained above then typedefed to the new TCB_t name
below to enable the use of older kernel aware debuggers. */
typedef tskTCB TCB_t;

```

Listing 5.3 – Descripteur de tâche



### 5.3.1 Liste de descripteurs de tâches

FreeRTOS gère des listes de tâches prêtes à devenir la tâche en cours. Les listes sont regroupées dans un tableau `pxReadyTasksLists`, chaque indice du tableau correspond à une priorité. Les recherches de la tâche de plus haute priorité sont ainsi très performantes. Il y a au moins une tâche disponible, la tâche IDLE, qui est une tâche de la plus basse des priorités automatiquement créée par l'ordonnanceur lors de son démarrage.

Les tâches en attente d'un délai sont déplacées dans une des listes, `pxDelayedTaskList` ou `pxOverflowDelayedTaskList`.

Les tâches suspendues sont déplacées dans `xSuspendedTaskList`.

Chaque tâche connaît la liste dans laquelle elle se trouve, cela permet d'aller interroger une tâche pour connaître son état.

Au niveau implémentation, les listes FreeRTOS sont des listes circulaires doublement chaînées. Le listing 5.4 montre les déclarations des listes utilisées. Les types utilisés pour les listes et les éléments des listes sont présentés dans les listings 5.5 et 5.6.

Les fonctions de FreeRTOS encapsulent les accès aux éléments des listes, il n'y aura jamais besoin d'accéder directement à ces listes.

```

/* Lists for ready and blocked tasks. -----*/
PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ]; /*< Prioritised
↳ ready tasks. */
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList; /*< Points to the
↳ delayed task list currently being used. */
PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList; /*< Points to the
↳ delayed task list currently being used to hold tasks that have overflowed the current
↳ tick count. */
PRIVILEGED_DATA static List_t xPendingReadyList; /*< Tasks that have been
↳ readied while the scheduler was suspended. They will be moved to the ready list when
↳ the scheduler is resumed. */

#ifdef INCLUDE_vTaskDelete == 1

PRIVILEGED_DATA static List_t xTasksWaitingTermination; /*< Tasks that have
↳ been deleted - but their memory not yet freed. */
PRIVILEGED_DATA static volatile UBaseType_t uxDeletedTasksWaitingCleanUp = (
↳ UBaseType_t ) 0;

#endif

#ifdef INCLUDE_vTaskSuspend == 1

PRIVILEGED_DATA static List_t xSuspendedTaskList; /*< Tasks that are
↳ currently suspended. */

#endif

```

Listing 5.4 – Listes internes de FreeRTOS

```
/*
 * Definition of the type of queue used by the scheduler.
 */
typedef struct xLIST
{
    listFIRST_LIST_INTEGRITY_CHECK_VALUE    /*< Set to a known value if
    ↪ configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    volatile UBaseType_t uxNumberOfItems;
    ListItem_t * configLIST_VOLATILE pxIndex;    /*< Used to walk through the list.
    ↪ Points to the last item returned by a call to listGET_OWNER_OF_NEXT_ENTRY (). */
    MiniListItem_t xListEnd;    /*< List item that contains the maximum possible
    ↪ item value meaning it is always at the end of the list and is therefore used as a
    ↪ marker. */
    listSECOND_LIST_INTEGRITY_CHECK_VALUE    /*< Set to a known value if
    ↪ configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
} List_t;
```

Listing 5.5 – Définition d’une liste FreeRTOS

```
struct xLIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE    /*< Set to a known value if
    ↪ configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    configLIST_VOLATILE TickType_t xItemValue;    /*< The value being listed. In most
    ↪ cases this is used to sort the list in descending order. */
    struct xLIST_ITEM * configLIST_VOLATILE pxNext;    /*< Pointer to the next ListItem_t
    ↪ in the list. */
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;    /*< Pointer to the previous
    ↪ ListItem_t in the list. */
    void * pvOwner;    /*< Pointer to the object (normally a TCB) that
    ↪ contains the list item. There is therefore a two way link between the object
    ↪ containing the list item and the list item itself. */
    struct xLIST * configLIST_VOLATILE pxContainer;    /*< Pointer to the list in which
    ↪ this list item is placed (if any). */
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE    /*< Set to a known value if
    ↪ configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
};
typedef struct xLIST_ITEM ListItem_t;    /* For some reason lint wants this as two
    ↪ separate definitions. */
```

Listing 5.6 – Définition d’un élément de liste FreeRTOS

## 5.4 Changement de contexte

Un changement de contexte, ou changement de tâche active, va se produire :

- A chaque fois qu’une tâche devient active, et est plus prioritaire que la tâche courante.

- Chaque fois que la tâche courante (qui est donc la plus prioritaire), passe en attente (d'un délai, ou d'un évènement).

Le test pour décider si il faut un changement de contexte va être lancé par l'ordonnanceur :

- Régulièrement, lors de l'interruption SysTick.

Mais ce changement peut aussi être provoqué :

- Depuis une interruption, quand un évènement a réveillé une tâche en attente qui a une priorité supérieure à la tâche en cours. Les commandes relatives à ce changement sont `portYIELD_FROM_ISR` et `portYIELD`. Ces fonctions sont dépendantes de l'implémentation (elles commencent par `port`). Pour le STM32F4, ces commandes génèrent manuellement une interruption, qui sera ensuite traitée comme l'interruption du SysTick.

## 5.5 Ordonnancement des tâches et préemption

### 5.5.1 Tick ou Tickless

FreeRTOS peut utiliser un mode de fonctionnement basse consommation, nommé **Tickless**. Ce mode est activé en définissant :

```
#define configUSE_TICKLESS_IDLE 1
```

dans le fichier de configuration `FreeRTOSConfig.h`.

Dans ce mode, la fonction `IDLE`, qui est appelée quand il n'y a pas de tâche plus prioritaire à effectuer, arrête l'ordonnanceur et passe le microcontrôleur en mode `SLEEP` pendant un temps calculé en fonction des réveils des futures tâches. Le microcontrôleur peut aussi être réveillé par une interruption.

Quand le mode **Tickless** n'est pas utilisé, l'interruption système (**SysTick**) se produit à intervalles réguliers.

Le mode **Tickless** peut engendrer du jitter sur les évènements périodiques.

Le mode **Tickless** n'est pas disponible sur tous les portages de **FreeRTOS**.

Nous utiliserons normalement le mode **Tick**. Cela n'empêche pas le microcontrôleur de passer en mode veille quand la tâche `IDLE` est activée. Mais le microcontrôleur sera réveillé lors de la prochaine interruption du `SystemTick`.

- En mode **Tick** :
  - Quand la tâche `IDLE` est activée, elle peut endormir le microcontrôleur ou consommer du temps.

- Le microcontrôleur est réveillé régulièrement par le **Systemtick**.
- Le microcontrôleur peut réagir aux interruptions (provenant de stimuli extérieur, de fin de conversion, de DMA, ...).
- En mode **Tickless** :
  - Quand la tâche **IDLE** est activée, elle endort le microcontrôleur pendant un temps calculé.
  - Le microcontrôleur peut réagir aux interruptions (provenant de stimuli extérieur, de fin de conversion, de DMA, ...).
  - L'heure interne est mise à jour au réveil.

### 5.5.2 Ordonnancement dans FreeRTOS

L'ordonnancement des tâches dans FreeRTOS respecte l'algorithme suivant :

- La tâche de plus haute priorité qui est prête à s'exécuter devient la tâche courante
- Si il y a plusieurs tâches de même priorité, elles vont s'exécuter séquentiellement (**Round Robin**). Chacune de ces tâches va avoir un intervalle de temps pour s'exécuter, puis la suivante de même priorité va s'exécuter. Ceci empêche les famines de tâches de même priorité.

Le choix des priorités pour les tâches est un problème complexe. Même en considérant que la priorité est fixée durant la durée de vie de l'application.

On trouve dans la littérature des méthodes de choix des priorités :

- RMS (Rate Monotonic Scheduling)
- EDF (Earliest Deadline First)
- ...

#### 5.5.2.1 RMS Rate Monotonic Scheduling

RMS est un algorithme à priorité constante. Les plus hautes priorités sont allouées aux tâches ayant les plus petites périodes. Cet algorithme est optimal uniquement dans la cas de tâches :

- indépendantes : elles ne doivent pas partager de ressources, de sémaphores, ...
- synchrones : elles ne réagissent pas sur des interruptions
- périodiques, avec une deadline inférieure ou égale à la période

L'algorithme RMS ne sera en général pas optimal dans une application où des événements sporadiques peuvent se produire. Et où des tâches devront interagir avec des ressources partagées.

### 5.5.2.2 EDF Earliest Deadline First

EDF est un algorithme à priorité variable. La plus haute priorité est allouée à la tâche ayant la deadline la plus proche.

Cet algorithme n'est pas implémenté dans **FreeRTOS**.

### 5.5.2.3 Autres algorithmes d'ordonnancement de tâches

On peut trouver d'autres algorithmes (LST, Sporadique, ...).

**FreeRTOS** ne les implémente pas, mais en cas d'utilisation d'un autre système d'exploitation temps réel il faudra se poser les questions suivantes :

- Que se passe t-il quand la charge processeur est proche de 100% ?
- Quel est le temps nécessaire à un meilleur ordonnancement, par rapport au temps d'exécution des tâches ?
- Quelle est la garantie (hard real time) du temps de latence dans tous les cas d'utilisation ?
- ...

## 5.6 Cycle de vie des tâches

Quand une tâche a été créée (voir en 5.6.1), elle ne peut plus avoir qu'un des états possibles :

**RUNNING** La tâche est en cours d'exécution

**NotRUNNING** La tâche n'est pas en cours d'exécution

**READY** Exécution possible (mais pas la plus haute priorité)

**SUSPENDED** En attente de commande **vTaskResume**

**BLOCKED** En attente :

- d'une durée
  - **vTaskDelay**
  - **vTaskDelayUntil**
- d'un événement de synchronisation
  - **xSemaphoreTake**
  - **xQueueReceive**
  - ...

La figure 5.1 représente le cycle de vie d'une tâche.

Durant l'exécution du programme, les tâches vont normalement alterner entre l'état **Running** et l'état **NotRunning**. Si ce n'est pas le cas il y a une famine de ressource. Cela peut être dû à une mauvaise architecture du programme, un mauvais choix de priorités, ....

Il ne peut y avoir qu'une et une seule tâche dans l'état **Running** à un instant donné. C'est le rôle de l'ordonnanceur, de déterminer quelle tâche doit devenir **Running**.

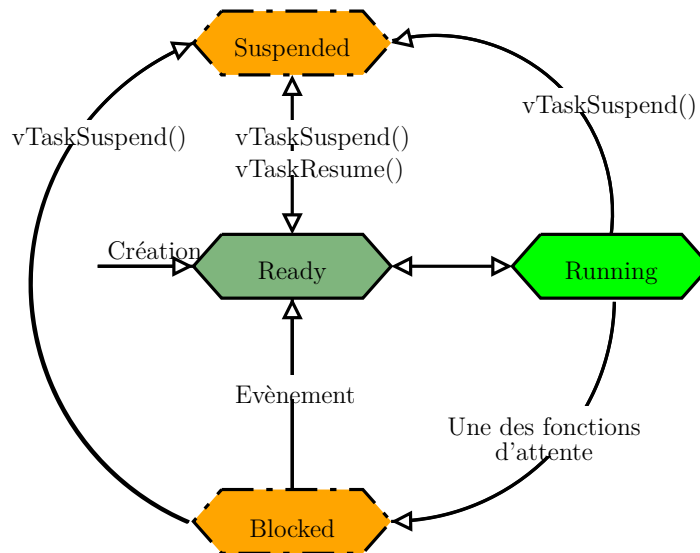


FIGURE 5.1 – Cycle de vie des tâches



### Choix de la tâche

Le choix de la tâche à exécuter est effectué par l'ordonnanceur (le scheduler), qui va régulièrement (voir le **Systick** en 5.2) vérifier quelle est la tâche de plus haute priorité qui est **READY**.

#### 5.6.1 Création de tâches

La tâche est l'élément de base d'un OS temps réel multitâches. L'ordonnanceur va gérer l'alternance des activations des tâches pour donner l'impression de simultanéité.

Une tâche va pouvoir se considérer comme étant seule à s'exécuter sur le microcontrôleur. Du point de vue du développeur, une tâche va être une fonction en langage C, ne se terminant pas. Et implémentée en général avec une simple boucle infinie.

Le listing 5.7 montre une implémentation d'une tâche :

- C'est une fonction
  - qui ne retourne rien : `void`

- qui accepte un seul paramètre : `void *pvParameters`
- Composée d'une boucle infinie, la tâche ne sera jamais terminée (même si elle peut être suspendue)
- Indépendante, la tâche ne se préoccupe pas de savoir si d'autres tâches sont présentes

```
1 void vSimpleTache( void *pvParameters )
2 {
3     for( ;; )
4     {
5         uint8_t uInput = GetInputStatus();
6         SetOutputStatus(uInput + 1);
7     }
8 }
```

Listing 5.7 – Une simple tâche

Pour que l'ordonnanceur sache qu'une fonction doit être considérée comme une tâche, et puisse être activée, il faut indiquer la création de cette tâche. Cela se fait avec la fonction `xTaskCreate` dont le prototype est le suivant :

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode, const char * const pcName,
↪ unsigned short usStackDepth, void *pvParameters, UBaseType_t uxPriority,
↪ TaskHandle_t *pxCreatedTask );
```

Listing 5.8 – Prototype du `xTaskCreate`

**pvTaskCode** C'est un pointeur vers la fonction qui servira de corps à la tâche. C'est à dire le nom de la fonction.

**pcName** Le nom de la tâche. Ceci est uniquement utilisé pour aider au débogage. Le nom de la tâche ne sert sinon à rien. Attention, la longueur du nom est limitée à `cnfigMAX_TASK_NAME_LEN`.

**usStackDepth** Représente la pile allouée pour cette tâche. Attention, cette valeur est en nombre d'éléments élémentaires et pas en nombre d'octets.

**pvParameters** Un pointeur sur des paramètres éventuels pour cette tâche. Cela peut être NULL.

**uxPriority** La priorité de cette tâche. La plage autorisée est 0 pour la plus basse priorité, et (`configMAX_PRIORITIES - 1`) pour la plus haute. La tâche IDLE a une priorité de 0.

**pxCreatedTask** Un pointeur sur un handle de la tâche (éventuellement NULL si cela n'est pas utilisé). Ce handle sert par la suite à modifier des paramètres de la tâche (la priorité, ou l'état suspendue par exemple)

Comme beaucoup de fonctions FreeRTOS, la valeur retournée par cet appel est :

**pdPASS** La tâche a été créée

**pdFAIL** La tâche n'as pas été créée, il n'y a pas de place mémoire suffisante pour stocker le descripteur de la tâche et la stack nécessaire.

Le listing 5.9 montre la création d'une tâche :

- Dont le corps est défini dans la fonction **vTaskChenillard**.
- Dont le nom (utile uniquement au trace et debugage) est **Chen**.
- Qui à 400\*32 bits de pile.
- Qui n'a pas de paramètre particulier (un pointeur NULL).
- Qui est plus prioritaire que la tâche de base IDLE.
- Qui va renvoyer son **handle** dans la variable **tChenillard**. Ce **handle** pourra servir aux fonctions de l'API (pour redéfinir la priorité, arrêter ou redémarrer la tâche par exemple).

```
TaskHandle_t tChenillard;
//...
if (pdPASS == xTaskCreate( vTaskChenillard, "Chen", 400, ( void * ) NULL,
↪ tskIDLE_PRIORITY+3, &tChenillard ))
{
    // la tâche a bien été créée
}
```

Listing 5.9 – Exemple de création de tâche avec **xTaskCreate**

### 5.6.2 Destruction de tâches

Suivant le modèle mémoire utilisé, quand il n'y a pas d'allocation et de libération mémoire, les destructions de tâches ne seront pas permises.

Dans ce cas là, il suffit de suspendre la tâche (**vTaskSuspend**) pour qu'elle ne soit plus traitée par l'ordonnanceur.

Le prototype de la fonction **vTaskDelete** est présenté en 5.10. Si de la mémoire doit être libérée après la destruction d'une tâche, cette libération ne se fera que dans la tâche IDLE. Il faut s'assurer que la tâche IDLE a des chances de pouvoir s'exécuter.



```
void vTaskDelete( TaskHandle_t pxTask );
```

Listing 5.10 – Prototype du `vTaskDelete`

### 5.6.3 Structure d'une tâche

Les tâches sont implémentées comme des fonctions contenant en général des boucles infinies. La tâche ne disparaît pas au cours de la vie de l'application. Elle est éventuellement suspendue (`vTaskSuspend`).

Le temps devant être partagé avec les autres tâches présentes, la tâche sera normalement en attente d'un événement déclencheur. Cela peut être :

- Un temps d'attente (pour les tâches effectuant des actions périodiques)
- Une attente d'un mécanisme de synchronisation (Sémaphore, notification, queue de communication, stream, ...)
- Ou en attente du fait de l'exécution d'une autre tâche plus prioritaire

Nos tâches devront toujours avoir le même principe :

- Une attente d'évènement déclencheur
- L'exécution du code relatif à l'évènement
- Et retour à l'attente

Le listing 5.11 montre des exemples de tâches qui suivent ces principes. Une tâche qui ne serait jamais arrêtée empêcherait le fonctionnement de toutes les tâches de priorités inférieures (famine).

```
void vTachePeriodique( void *pvParameters )
{
    while(1)
    {
        vTaskDelay(100); // attente 100 ticks
        action();
        // ...
    }
}

void vTacheAttenteEvenement( void *pvParameters )
{
    while(1)
    {
        xSemaphoreTake(xSemDoAction, portMAX_DELAY); // attente d'un évènement
        action();
        // ...
    }
}
```

Listing 5.11 – Structures de tâches

#### 5.6.4 Changement de priorité

Les tâches ont une priorité initiale, définie lors de la création de la tâche (voir en 5.6.1). Mais cette priorité peut évoluer suivant le contexte dans lequel l'application se trouve. Une fonction de FreeRTOS permet de relire la priorité d'une tâche, mais aussi d'en changer la priorité.

Les prototypes sont présentés sur le listing 5.12. Dans une tâche, il est possible de modifier ou de relire sa propre priorité, en utilisant NULL comme `TaskHandle_t` dans l'appel de la fonction.

Les priorités FreeRTOS sont des nombres entiers définies dans la plage `[0..configMAX_PRIORITIES-1]`.

0 est la plus basse priorité, celle qui est utilisée pour la tâche IDLE.

`configMAX_PRIORITIES-1` est la plus haute priorité. FreeRTOS gère un tableau de liste de tâches à une priorité donnée. Il faudra configurer `configMAX_PRIORITIES` (dans le fichier `FreeRTOSConfig.h`) à la valeur nécessaire pour l'application.

Le choix des priorités est un problème complexe, il faut identifier les fonctions critiques et s'assurer qu'il ne peut pas y avoir de famine pour les fonctions moins prioritaires.

Le changement de priorité de tâches est rarement utilisé.

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

Listing 5.12 – Prototype du `vTaskPrioritySet` et `uxTaskPriorityGet`

Il n'est possible de relire et de modifier les niveaux de priorités que si les fonctions sont autorisées dans `FreeRTOSConfig.h`.

```
#define INCLUDE_vTaskPrioritySet    1
#define INCLUDE_uxTaskPriorityGet  1
```

### 5.6.5 La tâche IDLE

La tâche IDLE est créée automatiquement par FreeRTOS. Elle a la priorité la plus basse (0), c'est la tâche qui doit s'exécuter quand il n'y a pas d'autres tâches plus prioritaires qui sont prêtes (à l'état `READY`).

Si des tâches utilisateurs ont la même priorité que la tâche IDLE, il faut configurer FreeRTOS avec

```
#define configIDLE_SHOULD_YIELD    1
```

pour éviter que la tâche IDLE ne prenne tout le temps alloué aux tâches de priorité 0.

Si la fonction de destruction de tâches est utilisée (`vTaskDelete`), il faut s'assurer que la tâche IDLE pourra être appelée, car c'est elle qui se charge de l'élimination des tâches détruites.

Il n'est pas possible de modifier la fonction IDLE, mais il est possible de définir un hook (crochet) pour être notifié quand la fonction IDLE est appelée.

```
#define configUSE_IDLE_HOOK        1
```

Dans ce cas là il faut fournir une fonction :

```
void vApplicationIdleHook( void );
```

Un appel régulier de la fonction IDLE, et de la fonction `vApplicationIdleHook` est un signe de bon fonctionnement de l'application. Cela indique qu'il n'y a pas une tâche qui monopolise toutes les ressources. Il est courant d'avoir une utilisation de 90% des ressources du microcontrôleur affecté à la tâche IDLE.



# 6

## Gestion de la mémoire

*Ce chapitre traite de la gestion mémoire avec FreeRTOS, et comment les problèmes d'allocation/libération mémoire ont été résolus.*

*“As you get older three things happen. The first is your memory goes, and I can't remember the other two.”*

- Norman Wisdom,

### 6.1 Problème d'allocation et contraintes de l'embarqué

La gestion mémoire est un problème critique dans un système embarqué. Le langage C est assez permissif pour l'utilisation des allocations et désallocations mémoire. Il n'y a pas de protection contre une erreur de codage, et les causes de plantage de l'application peuvent être diverses :

- Trop d'allocation sans libération, il n'y a plus de place mémoire disponible.
- Utilisation d'une zone mémoire libérée.
- Utilisation d'un pointeur NULL.
- Fragmentation de la mémoire, les allocations de zone mémoire ne s'exécutent pas en un temps fixé.
- Les fonctions d'allocation ne sont pas toujours thread-safe.
- ...

On essaiera de ne pas utiliser d'allocation dynamique, mais de définir statiquement toutes les zones mémoires nécessaires à l'application, sans avoir à faire d'allocation et de libération.

## 6.2 Gestionnaires mémoire

FreeRTOS fournit plusieurs gestionnaires mémoire qui encapsulent les fonctions d'allocation et de libération. Les gestionnaires sont fournis dans un dossier `MemMang`, et sont nommés `heap_1.c..heap_5.c`. Il est possible de mixer les gestionnaires mémoires, et il est même possible de réécrire le sien.

**heap\_1.c** Est un gestionnaire simple, qui ne permet pas la libération mémoire. Il est utilisé dans la majorité des cas, car un système embarqué connaît toutes les tâches qu'il devra exécuter. Il est alors possible d'allouer toutes les ressources nécessaires au début de l'application. Il n'y a pas besoin ensuite de nouvelles allocations de mémoire.

**heap\_2.c** Permet les libérations de mémoire, mais ne protège pas contre la fragmentation mémoire. Ce gestionnaire n'est pas déterministe, mais est plus efficace que le gestionnaire de mémoire standard du langage C.

**heap\_3.c** Implémente juste une protection au dessus des fonctions standards d'allocation et de libération (`malloc` et `free`). Il peut être utilisé quand ces fonctions sont présentes, mais ne sont pas protégées contre les accès concurrents.

**heap\_4.c** Est similaire à `heap_2.c`, mais tente d'éviter les problèmes de fragmentation mémoire.

**heap\_5.c** Est similaire à `heap_4.c`, mais permet d'utiliser plusieurs zones mémoires, au lieu d'un seul bloc pour les autres gestionnaires.

Le fichier `FreeRTOSConfig.h` contient les paramètres de configuration `configMINIMAL_STACK_SIZE` et `configTOTAL_HEAP_SIZE`.

```
#define configMINIMAL_STACK_SIZE    ( ( unsigned short ) 100 )
#define configTOTAL_HEAP_SIZE       ( ( size_t ) ( 30 * 1024 ) )
```

Suivant le gestionnaire mémoire utilisé, une zone mémoire `ucHeap` sera statiquement allouée et utilisée par `FreeRTOS`.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

`configMINIMAL_STACK_SIZE` est la taille utilisée pour la fonction `IDLE` qui est automatiquement créée par `FreeRTOS` quand l'ordonnanceur est démarré.

## 6.3 Erreurs liées à la mémoire

### 6.3.1 Erreur d'allocation

Le fichier de configuration de FreeRTOS contient des définitions permettant de changer des réglages de fonctionnement.

En définissant :

```
#define configUSE_MALLOC_FAILED_HOOK 1
```

Il faut aussi fournir une fonction `vApplicationMallocFailedHook` qui sera appelée si un problème d'allocation mémoire se produit. Le listing 6.1 montre un exemple de ce type de fonction.

En général, elles permettent de mettre un point d'arrêt, ce qui permet à un debugger de venir analyser la raison du problème. La boucle infinie permet d'éviter que le programme ne continue.

```
1 void vApplicationMallocFailedHook( void )
2 {
3     taskDISABLE_INTERRUPTS(); // plus d'ordonnanceur
4     for( ;; ); // boucle infinie
5 }
```

Listing 6.1 – Exemple de `vApplicationMallocFailedHook`

La fonction `xPortGetFreeHeapSize()` permet de lire la taille mémoire restante. C'est à dire la partie `configTOTAL_HEAP_SIZE` qui n'est pas encore utilisée.

```
size_t xPortGetFreeHeapSize( void );
```

### 6.3.2 Erreur de stack

Une erreur de pile peut se produire lors de l'exécution d'une fonction, si il y a plusieurs contextes imbriqués, lors d'appels récursifs, ... La taille de la pile est définie lors la création de la tâche.

FreeRTOS fournit la fonction `uxTaskGetStackHighWaterMark` dont le prototype est :

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

Cette fonction renvoie la taille minimum atteinte par la pile de cette tâche. Quand la pile est proche de 0, il y a un risque. Quand la pile est trop grosse de la mémoire est perdue (non utilisée).

La fonction `uxTaskGetStackHighWaterMark` est en général utilisée lors du développement, pour régulièrement analyser les tâches en cours d'exécution.

FreeRTOS permet d'utiliser un mécanisme de vérification de la pile. En définissant :

```
#define configCHECK_FOR_STACK_OVERFLOW 0, 1 ou 2
```

Il est possible d'effectuer du contrôle sur la pile.

Il faut fournir la fonction `vApplicationStackOverflowHook` définie par le prototype suivant :

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask, signed char  
↪ *pcTaskName );
```

Grâce aux paramètres `pxTask` et `pcTaskName` il est possible de retrouver la tâche qui a créé le problème de pile.

Suivant la valeur de `configCHECK_FOR_STACK_OVERFLOW` :

- 0 Pas de vérification de la pile, la fonction `vApplicationStackOverflowHook` n'est pas nécessaire et ne sera pas appelée.
- 1 Il y a une vérification à chaque changement de contexte que le pointeur de pile est bien dans la zone prévue. Sinon `vApplicationStackOverflowHook` est appelée.
- 2 Cette option permet d'effectuer les vérifications, et ajoute un test de non corruption de la pile entre les changements de contexte. Un pattern connu est stocké dans la pile. Si il est modifié, cela signifie qu'un débordement a eu lieu, et la fonction `vApplicationStackOverflowHook` est alors appelée.

Nous verrons la mise en œuvre de ces fonctions lors des exercices.



# 7

## Ressources critiques, MutEx, synchro, ...

*Ce chapitre traite de la protection des ressources critiques, et des fonctions de synchronisation entre les tâches.*

*“If debugging is the process of removing software bugs, then programming must be the process of putting them in.”*

- Edsger Dijkstra,

## 7.1 Gestion des ressources

### 7.1.1 Exclusion mutuelle

En utilisant un système temps réel multitâches, nous nous trouvons au cœur des risques liés aux partages de ressources.

L’ordonnanceur va allouer du temps à chaque tâche, mais n’a aucune connaissance des actions effectuées par chaque tâche.

Pour tout accès non atomique à une ressource, il y a un risque que la ressource soit laissée dans un certain état par une tâche interrompue, et que cet état soit modifié par une autre tâche. De retour dans la tâche d’origine, cela peut occasionner des problèmes d’accès à la ressource.

Ces problèmes vont pouvoir se rencontrer chaque fois qu’une ressource va être partagée par :

- Des tâches
- Des tâches et des interruptions

Voici quelques exemples de ce phénomène :

#### L’accès à une EEPROM par 2 tâches

L’accès à une EEPROM s’effectue en général par 2 accès :

- Définition de l'adresse d'écriture
- Définition de la data à écrire

Si un changement de tâche se produit entre ces 2 opérations, l'adresse d'écriture peut être changée avant le retour dans la tâche d'origine. L'écriture se produira à un emplacement non désiré.

### La remise à 0 d'une variable dans une interruption

Si l'on considère un compteur qui est incrémenté dans une tâche, et remis à 0 par une interruption comme dans le code du listing 7.1. Même si l'interruption est correctement appelée, il n'est pas sûr que le compteur soit bien remis à 0.

L'interruption peut se produire à tout moment, si elle se produit pendant l'incrémementation (qui n'est pas atomique), le contenu de la variable `uMonCompteurCritique` n'est pas connu.

```

1  volatile uint32_t uMonCompteurCritique = 0;
2
3  void __interrupt MonInterruption()
4  {
5      uMonCompteurCritique = 0;
6  }
7
8  ...
9
10 void MaFonctionQuiCompte(void)
11 {
12     while(1)
13     {
14         uMonCompteurCritique++;
15     }
16 }
```

Listing 7.1 – Problème d'accès aux ressources

### L'accès à une liaison série par 2 tâches

Cet exemple est le plus facile à mettre en œuvre pour vérifier les effets d'un mauvais accès à un périphérique et cela nous servira de fils rouge dans les exemples.

Si l'on considère des tâches qui accèdent à la même liaison série, du fait des changements de contexte, les données envoyées peuvent se retrouver mixées.

Ce soucis et le moyen de le contourner seront présentés dans les exercices.



## Partage de ressources

Il y aura *toujours* un risque quand une ressource est partagée entre des tâches, ou entre des tâches et des interruptions.

FreeRTOS va nous fournir des méthodes pour protéger les accès aux ressources, ce mécanisme d'*exclusion mutuelle* va garantir qu'une tâche a un accès exclusif à une ressource.

Mais la meilleure méthode d'exclusion mutuelle est de définir une architecture qui garantisse que les ressources ne sont pas partagées.

### 7.1.1.1 Sections critiques

Une section critique va être une partie du code qui ne doit pas pouvoir être interrompue.

Ceci est évidemment contradictoire avec le fait que des tâches plus prioritaires ne doivent pas attendre la fin d'une tâche moins prioritaire.

FreeRTOS offre plusieurs moyens pour gérer les sections critiques, en continuant à garantir au mieux la préemption des tâches prioritaires et le bon fonctionnement des interruptions.

### 7.1.1.2 Protection avec des fonctions de l'ordonnanceur

**En arrêtant les interruptions** L'arrêt des interruptions entraîne qu'il n'y a plus de changement de contexte, mais il n'y a pas non plus d'autres interruptions. Quand il n'y a pas de système d'exploitation temps réel, cette méthode est souvent la seule possible quand des variables sont partagées entre le programme principal et une interruption.

Les fonctions de FreeRTOS pour inhiber et réautoriser les interruptions sont :

```
void taskENTER_CRITICAL( void );
void taskEXIT_CRITICAL( void );
```



## taskENTER\_CRITICAL taskEXIT\_CRITICAL

- Ces fonctions sont **obligatoirement** appairées.
- Les fonctions de synchronisation FreeRTOS ne sont plus disponibles.
- Il y a un risque de perte d'évènements, si des interruptions se produisent durant l'arrêt.

**En arrêtant l'ordonnanceur** L'arrêt de l'ordonnanceur seulement empêche le changement de contexte entre les tâches. Des interruptions peuvent encore être traitées mais les fonctions de synchronisations, de protections et d'attente de FreeRTOS ne sont plus valides.

- Les interruptions continuent d'arriver
- Mais sans ordonnanceur, il n'y a peut être pas de code pour les gérer (on peut manquer des évènement au final)
- Les tâches de priorités élevées ne sont pas exécutées non plus

Les fonctions de FreeRTOS utilisées sont :

```
void vTaskSuspendAll( void );  
BaseType_t xTaskResumeAll( void );
```



#### **vTaskSuspendAll xTaskResumeAll**

- Ces fonctions sont **obligatoirement** appairées
- Les fonctions qui changent le contexte de FreeRTOS ne sont plus disponibles
- FreeRTOS maintient le bon compte de ticks durant les arrêts

### **7.1.1.3 Protection avec un objet de synchronisation**

**Le MutEx, jeton d'accès exclusif** Le MutEx<sup>1</sup> va être un objet partagé par des tâches ou des interruptions. L'accès atomique au mutex est assuré par FreeRTOS.

Le mutex doit être pris, puis rendu.

Les avantages du mutex :

- Le concept est simple, celui qui possède le mutex accède à la ressource
- Très efficace
- FreeRTOS fournit un grand nombre de fonctions relatives aux mutex, pour attendre qu'il devienne disponible (avec timeout), appels récursifs, ...

Mais il y a des risques :

- Rien n'empêche de ne pas respecter le mutex
- Si le mutex n'est pas rendu, la ressource n'est plus disponible
- Tout dépend de la discipline du développeur

---

1. Mutual Exclusion

### Les mutex dans la littérature

Vous trouverez souvent les notations suivantes :

Prise de mutex / Libération de mutex

- P V (notation originale de Dijkstra)
- wait signal
- acquire release
- pend post
- down up
- procure vacate
- ...

Avec FreeRTOS, les commandes principales sont :

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t
↪ xTicksToWait );
BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Les Mutex, et leurs utilisations sont décrits au chapitre 7.1.2.

## 7.1.2 Les mutex

### 7.1.2.1 Mutex ou sémaphore

Les Mutex sont des sémaphores binaires. C'est à dire des objets pouvant prendre 2 états :

- Disponible
- Non disponible

Les Mutex interviennent dans les scénarios de protection d'une ressource. Chaque partie du programme qui nécessite la ressource accepte de demander l'autorisation au préalable (en attendant que le Mutex soit disponible). A la fin de l'utilisation de la ressource, il est nécessaire de rendre le Mutex.

Les sémaphores interviennent dans les scénarios de synchronisation. Une tâche attend un sémaphore. Quand le sémaphore est donné par une autre tâche (ou une interruption), la première tâche est débloquée (elle se synchronise).

L'état initial à la création d'un Mutex est **disponible**. La première tâche qui requiert un accès à la ressource peut y accéder. Un Mutex ne peut être rendu que par la tâche qui l'a pris.

L'état initial d'un sémaphore est **non disponible**. La première tâche qui attend un sémaphore sera bloquée. Un sémaphore est dédié à la synchronisation entre tâches et/ou interruptions. Il peut être pris et rendu sans contrainte.

Pour les sémaphores et MutEx, FreeRTOS garanti que les tâches les plus prioritaires sont servies en premier, et qu'en cas de priorité identique, la tâche qui attend depuis le plus longtemps sera servie.

#### 7.1.2.2 Récuratif ou pas

Si une tâche possède un MutEx, et que pour une raison d'algorithme elle demande à nouveau le même MutEx, elle va se retrouver bloquée par elle-même. Sans aucune chance de se retrouver débloquée.

Pour résoudre ce problème, il est possible d'utiliser des MutEx récursifs. Les commandes à utiliser sont :

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
BaseType_t xSemaphoreTakeRecursive( SemaphoreHandle_t xSemaphore, TickType_t
↳ xTicksToWait );
BaseType_t xSemaphoreGiveRecursive( SemaphoreHandle_t xSemaphore );
```

#### 7.1.2.3 Le problème de l'inversion de priorité

L'utilisation de mutex pour protéger une section critique peut entraîner un dysfonctionnement des règles du système d'exploitation temps réel : *Une tâche de basse priorité peut empêcher une tâche de haute priorité de s'exécuter.*

On considère un système avec 3 tâches de priorités  $T3 > T2 > T1$ .

T1 et T3 partagent une ressource, et utilisent donc un mutex pour protéger l'accès à cete ressource.

- T1 possède le mutex
- T3 préempte T1 mais se bloque (mutex)
- T1 reprend son fonctionnement
- T2 préempte T1
- T2 continue à s'exécuter
- T3 **pourtant prioritaire** reste **bloquée**
- ...

La figure 7.1 représente cet enchaînement de tâches.

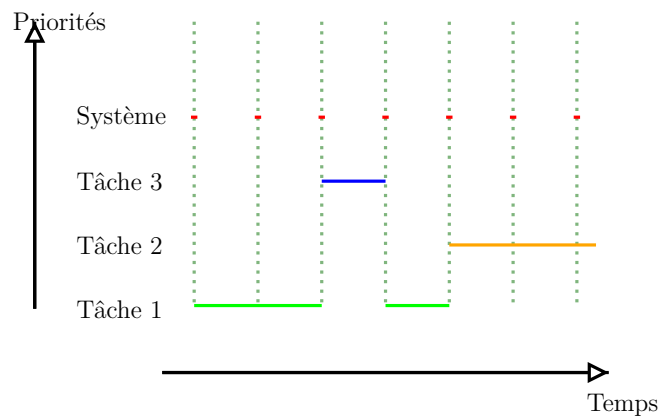


FIGURE 7.1 – Problème d’inversion de priorités sans héritage

Dans cet exemple : tant que la tâche T2 s’exécute, la tâche T1 ne libère pas le mutex, ce qui entraîne que le tâche T3 pourtant plus prioritaire reste bloquée.

FreeRTOS gère ce problème en augmentant la priorité de la T1. Sur la figure 7.3, on peut voir que le fait d’avoir augmenté la priorité de la tâche T1 au niveau de la tâche T3 a permis de ne pas rester bloqué par la tâche T2.

C’est ce problème qui est arrivé à Mars Pathfinder (en 1997).



FIGURE 7.2 – Mars pathfinder

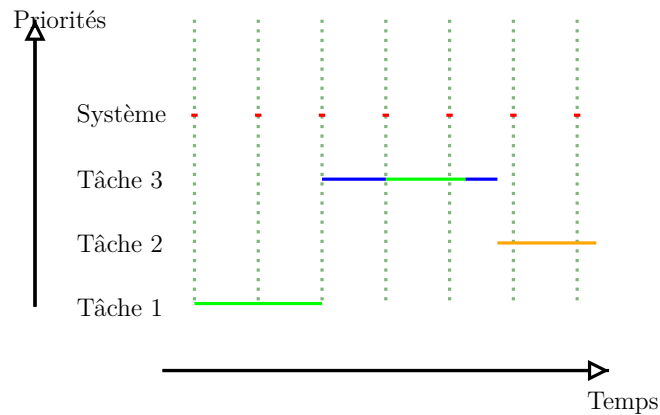


FIGURE 7.3 – Problème d'inversion de priorités avec héritage

#### 7.1.2.4 Héritage de priorité

Les mutex **FreeRTOS** implémentent un héritage des priorités. Cela permet de s'affranchir en parti des problèmes d'inversions de priorités (vu en 7.1.2.3).

La priorité d'une tâche va être augmentée jusqu'à la priorité réclamant un mutex possédé par cette tâche. Ce mécanisme n'est pas déterministe, et peut avoir des effets de bord.

#### 7.1.2.5 Les Deadlock

Le phénomène de Deadlock (ou étreinte mortelle) peut se produire lorsque l'ordre de prise de plusieurs **MutEx** n'est pas identiques dans plusieurs tâches.

Le listing 7.2 montre une mauvaise utilisation des **MutEx**. Si la **tacheB** interrompt la **tacheA** car elle devient plus prioritaire, suivant l'endroit de l'intrruption cela peut provoquer un blocage complet des deux tâches.

Enchainement pour obtenir un blocage :

- La **tacheA** s'exécute.
- La **tacheB** interrompt la **tacheA** régulièrement (elle la préempte).
- Cette interruption peut se produire à tout endroit de la **tacheA**.
- Si l'interruption de la **tacheA** se produit entre les deux prises de **MutEx** :
  - La **tacheA** prend le **MutEx1**.
  - La **tacheA** est interrompue par la **tacheB**.
  - La **tacheB** prend le **MutEx2**.
  - La **tacheB** est bloquée car elle ne peut pas prendre le **MutEx1**.
  - La **tacheB** étant bloquée, la **tacheA** peut continuer à s'exécuter.



- Mais se retrouve bloquée aussi car le **MutEx2** n'est pas disponible.
- Les deux tâches sont bloquées. Il y a Deadlock.

```
void tacheA()
{
    while(1)
    {
        P(mutex1);
        P(mutex2);

        //... execute une action sur des ressources

        V(mutex2);
        V(mutex1);
    }
}
void tacheB()
{
    while(1)
    {
        P(mutex2);
        P(mutex1);

        //... execute une action sur des ressources

        V(mutex1);
        V(mutex2);
    }
}
```

Listing 7.2 – Phénomène de Deadlock

Ce problème peut être évité en s'assurant que les **MutEx** seront toujours pris dans le même ordre (et en les rendant dans l'ordre contraire). Mais ceci n'est pas toujours simple à vérifier, en particuliers quand des tâches appellent des fonctions qui elles mêmes appellent des fonctions qui prennent et libèrent des **MutEx**.

Quand cela est possible, on peut utiliser un **MutEx** supplémentaire, qui va protéger l'accès à tous les autres **MutEx**.

### 7.1.3 Le Gatekeeper

L'utilisation de Gatekeeper permet de protéger l'accès à une ressource en s'affranchissant des problèmes liés à l'inversion de priorité et aux deadlock.

Il faut implémenter une tâche Gatekeeper par ressource à protéger. Cette tâche sera la seule et unique ayant le droit d'accéder à la ressource.

Les Gatekeeper utilisent en général une queue de communication (voir en 7.3) comme mécanisme de communication. Il n'y a pas de soucis d'exclusion mutuelle du fait de l'unicité de l'accès à la ressource.

- La technique du gatekeeper consiste à dédier l'accès exclusif d'une ressource à une tâche.
- C'est la **seule** partie du programme qui accède à la ressource.
- Les accès sont séquentiels, en général en réponse à des commandes reçues sur une queue de communication.

Pour l'implémentation d'une tâche Gatekeeper, il faut définir quelle va être la structure de données transmises à la tâche pour effectuer une action. Cela peut être un type simple (`int`, `char`, ...) mais en général ce sera une structure pouvant encapsuler un grand nombre d'informations et de paramètres.

Le listing 7.3 montre une structure de transfert de données qui peut être utilisée par une tâche Gatekeeper gérant l'accès à une ressource de type "Ensemble de Led 3 couleurs". Les paramètres de la structure sont :

- `uPosition` Le numéro de la led
- `uRed`, `uGreen`, `uBlue` des booléens représentant les 3 composantes lumineuses de la led.

```

1 typedef struct
2 {
3     uint8_t uPosition;
4     bool uRed;
5     bool uGreen;
6     bool uBlue;
7 } ledInfo_t;
```

Listing 7.3 – Structure pour un Gatekeeper

La tâche Gatekeeper est créée comme toutes les autres tâches, avec l'instruction `xTaskCreate`.

```

xTaskCreate(vTaskLedGateKeeper, "LGK", 400, (void *)NULL, tskIDLE_PRIORITY+1,
↪ NULL);
```

Qui pourra être définie comme dans le listing 7.4. Le cycle de vie de la tâche Gatekeeper est :

- La tâche est bloquée (sans timeout) en attente de réception d'un élément de type `ledInfo_t`.
- Quand un élément arrive il est décodé et exécuté.
- La tâche se remet en attente.

```

1 void vTaskLedGateKeeper(void *pParameters)
2 {
3     ledInfo_t ledInfo;
4     for(;;)
5     {
6         // attente d'une commande qui doit s'exécuter sur les leds
7         xQueueReceive(xQueueLedInfo, &ledInfo, portMAX_DELAY);
8         switch(ledInfo.uPosition)
9         {
10            case 0:ledH(ledInfo.uRed, ledInfo.uGreen, ledInfo.uBlue);break;
11            case 1:ledB(ledInfo.uRed, ledInfo.uGreen, ledInfo.uBlue);break;
12            case 2:ledG(ledInfo.uRed, ledInfo.uGreen, ledInfo.uBlue);break;
13            case 3:ledD(ledInfo.uRed, ledInfo.uGreen, ledInfo.uBlue);break;
14        }
15    }
16 }

```

Listing 7.4 – Tâche Gatekeeper

L'envoi de données à la tâche Gatekeeper pourra être implémenté comme dans le listing 7.5.

Cette méthode a l'avantage que :

- Il n'y a pas d'attente lors de l'envoi d'informations vers la queue de communication.
- Plusieurs éléments peuvent être stockés dans la queue de communication.

Mais il peut y avoir quelques inconvénients :

- On ne maîtrise pas le temps entre l'envoi dans la queue de communication et la réelle prise en compte.
- Les queues de communication peuvent consommer beaucoup de mémoire.

```

1 ledInfo_t ledInfo;
2 ledInfo.uPosition = 0;
3 ledInfo.uRed = r;
4 ledInfo.uGreen = g;
5 ledInfo.uBlue = b;
6 xQueueSend(xQueueLedInfo, &ledInfo, 0); // 0=>arbitraire, si plus de place, on
   ↪ oublie

```

Listing 7.5 – Envoi de données vers le Gatekeeper

## 7.2 Primitives de synchronisation

### 7.2.1 Sémaphores binaires

Les sémaphores permettent d'effectuer un transfert d'information entre des tâches et/ou entre des tâches et des interruptions. L'information transmise est ici minimaliste, nous transmettons juste l'état du sémaphore.

Une application classique des sémaphores binaires est la synchronisation entre une interruption et une tâche.

En général dans une application :

- Nous ne voulons pas rater d'interruption.
- Mais le code lié à une interruption est peut être d'une priorité inférieure à la tâche en cours.

Cela se résout en différant l'interruption (voir aussi en 8.1.8). Le listing 7.6 montre une tâche bloquée, en attente d'un sémaphore.

Durant l'attente, la tâche ne consomme pas de ressources. L'attente est infinie en utilisant la constante `portMAX_DELAY`.

Le listing 7.7 montre un code qui peut être déclenché sur une interruption. Quand cela se produit, la tâche en cours est interrompue, et le code `vMonISR_Handler` s'exécute.

Le sémaphore `xSemaphoreGiveFromISR` est donné, ce qui va réveiller la tâche `MaTacheQuiAttend`. La variable `xHigherPriorityTaskWoken` reçoit `pdFALSE` ou `pdTRUE` si une tâche de priorité plus élevée que la tâche courante a été réveillée par ce sémaphore. La fonction `portYIELD_FROM_ISR` va effectuer un changement de contexte dans ce cas là. Cela permet de minimiser la latence<sup>2</sup>.

```

1 void MaTacheQuiAttend( void * pvParameters )
2 {
3     while(1)
4     {
5         // Bloquons en attente d'un évènement
6         if( xSemaphoreTake( xSemaphore, portMAX_DELAY ) == pdTRUE )
7         {
8             // action synchronisée avec l'interruption
9             // la latence est minimale
10        }
11    }
12 }
```

Listing 7.6 – Tâche qui attend une interruption

2. Le temps entre l'évènement déclencheur et la première ligne de code de gestion de l'évènement.

```

1 // Du code appelé par une interruption (Timer, GPIO, fin de conversion, ...)
2 void vMonISR_Handler( void * pvParameters )
3 {
4     static signed BaseType_t xHigherPriorityTaskWoken;
5
6     xHigherPriorityTaskWoken = pdFALSE;
7     xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );
8
9     //Changement de contexte
10    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
11 }

```

Listing 7.7 – Utilisation de sémaphore de synchronisation depuis une interruption

### 7.2.1.1 Cycle de vie d'un sémaphore

Un sémaphore est généralement créé au début de l'application. Le handle du sémaphore est global, ou passé en paramètre pour que toutes les fonctions qui ont besoin d'y accéder le puissent.

Il y aura en général une variable globale :

```
SemaphoreHandle_t xSemSynchroBouton = NULL;
```

Et une déclaration dans le corps du programme :

```
xSemSynchroBouton = xSemaphoreCreateBinary();
```

Les fonctions de création sont de la forme `xSemaphoreCreatexxxx`, le `xxxx` représente les différentes formes des sémaphores (MutEx, binaires, compteurs, récursifs, ...).

Les tâches peuvent ensuite attendre la présence du sémaphore ou signaler un sémaphore.

### 7.2.1.2 Signaler un sémaphore

Le signalement d'un sémaphore se fait avec une des fonctions de la famille `xSemaphoreGive`.

- `xSemaphoreGiveFromISR` depuis une interruption
- `xSemaphoreGive` depuis une tâche

Ces fonctions peuvent renvoyer `pdFALSE` en cas d'erreur ou `pdTRUE` dans le cas où le sémaphore a bien été donné.

### 7.2.1.3 Attendre un sémaphore

L'attente d'un sémaphore se fait avec une des fonctions de la famille `xSemaphoreTake`. Ces fonctions sont bloquantes, avec un timeout, sauf celle qui est dédiée aux interruption (`FromISR`) qui ne peut pas être bloquante.

- `xSemaphoreTakeFromISR` depuis une interruption
- `xSemaphoreTake` depuis une tâche

Dans la fonction `xSemaphoreTake`, le paramètre `xTicksToWait` est un temps en ticks, ce qui permet de limiter le temps d'attente.

- `xTicksToWait` peut être égal à la constante `portMAX_DELAY`, ce qui entraîne une attente sans limite. Quand `xTicksToWait` est `portMAX_DELAY`, ce n'est pas la peine de tester le retour de la fonction qui est automatiquement `pdTRUE`. Dans les autres cas, il faut tester le retour de la fonction pour vérifier si l'on a effectivement reçu un signalement du sémaphore ou un timeout.
- La macro `pdMS_TO_TICKS()` permet d'effectuer une conversion de ticks en ms.

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t
↪ xTicksToWait );
```

Le listing 7.6 montre un exemple d'attente de sémaphore.

## 7.2.2 Evènements

Les évènements (au sens **FreeRTOS**) sont des groupes de bits sur lesquels il est possible d'effectuer des opérations sûres, que les fonctions soient appelées depuis une tâche ou depuis une interruption (avec les variantes `...FromISR`). On parlera aussi d'évènements pour parler des interruptions ou des exceptions. Mais ce chapitre traite uniquement des **EventGroup**. Les interruptions seront traitées en 8.1.8.

L'utilisation de ces fonctions évite la mécanique habituelle de protection des variables :

1. Arrêt des interruptions
2. Accès aux variables
3. Autorisation des interruptions

Le niveau d'abstraction fourni par **FreeRTOS** permet de s'affranchir de la protection de ces variables en encapsulant tous les appels dans des fonctions de haut niveaux.

Avec des groupes de bits il est ainsi possible de :

- Définir ou lire des bits dans un ensemble

- Attendre que 1 ou plusieurs bits soient présents, ou que 1 bit parmi plusieurs soit présent.
- Définir un bit et attendre d'autres bits en une seule étape (rendez-vous de tâches).

### 7.2.2.1 Cycle de vie d'un évènement

Un groupe de bits doit être créé avant utilisation.

```
EventGroupHandle_t xEventGroupCreate( void );
```

La variable doit être globale si elle doit servir dans plusieurs endroit du programme.

### 7.2.2.2 Signaler un évènement

Le signalement d'un évènement se fait grâce à la fonction :

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const  
↪ EventBits_t uxBitsToSet );
```

Plusieurs évènements (appartenant au même groupe d'évènements) peuvent être signalés simultanément.

Si il y a plusieurs tâches qui attendent un des évènements signalé, c'est la tâche de plus haute priorité qui sera servie en premier.

### 7.2.2.3 Attendre un évènement

L'attente d'un évènement se fait grâce à la fonction :

```
EventBits_t xEventGroupWaitBits( const EventGroupHandle_t xEventGroup, const  
↪ EventBits_t uxBitsToWaitFor, const BaseType_t ClearOnExit, const BaseType_t  
↪ xWaitForAllBits, TickType_t xTicksToWait );
```

La tâche va être bloquée jusqu'à l'obtention d'un ou plusieurs bits (suivant la valeur de `xWaitForAllBits`) du masque `uxBitsToWaitFor`. Les bits peuvent être éventuellement être remis à 0 (suivant la valeur de `ClearOnExit`).

### 7.2.2.4 Synchronisation d'un évènement

La synchronisation d'évènement se fait grâce à la fonction :

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup, const EventBits_t  
↪ uxBitsToSet, const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait );
```

En une seule fonction, les bits `uxBitsToSet` sont définis, et la tâche attend les `uxBitsToWaitFor` durant `xTicksToWait` ticks. Ce mécanisme appelé **rendez-vous** permet de synchroniser plusieurs tâches entre elles, généralement pour garantir que le système démarre quand toutes les initialisations sont effectuées.

Le projet d'exemple `basicProjectFreeRTOSwButton` contient un rendez-vous au début de chaque tâche.

Le principe est que chaque tâche devant participer au rendez-vous indique qu'elle est prête (bit de présence), et attend que l'ensemble des tâches soient prêtes.

Au début de chaque tâche, on va retrouver ce type de code :

```
xEventGroupSync(xTasksReady, BIT_TASK_DEBUG, BIT_ALL_TASKS, portMAX_DELAY);
```

Avec :

- `xTasksReady` un `EventBits_t` créé par la commande `xEventGroupCreate()`.
- `BIT_TASK_DEBUG` est le bit de présence lié à la tâche.
- `BIT_ALL_TASKS` est l'ensemble des bits des tâches participant au rendez-vous.

```
// bits de rendez-vous pour les tâches
#define BIT_TASK_LED      (1<<0)
#define BIT_TASK_BUTTON (1<<1)
#define BIT_TASK_CONFIG (1<<2)
#define BIT_TASK_DEBUG   (1<<3)

#define BIT_ALL_TASKS (BIT_TASK_LED | BIT_TASK_BUTTON | BIT_TASK_CONFIG |
↪ BIT_TASK_DEBUG)
```

## 7.3 Communication entre tâches et Queues de communication

Les queues de communication servent (entre autre) à résoudre des problèmes du genre :

1. Un **producteur** génère des données :
  - Il a une périodicité de génération irrégulière
  - Il ne doit pas être bloqué
  - Les données ne doivent pas se perdre
2. Un **consommateur** utilise les données :
  - L'utilisation des données prend un temps important (relativement)
  - Pendant l'utilisation des données le producteur continue à ... produire



### 7.3.1 Un cas simple

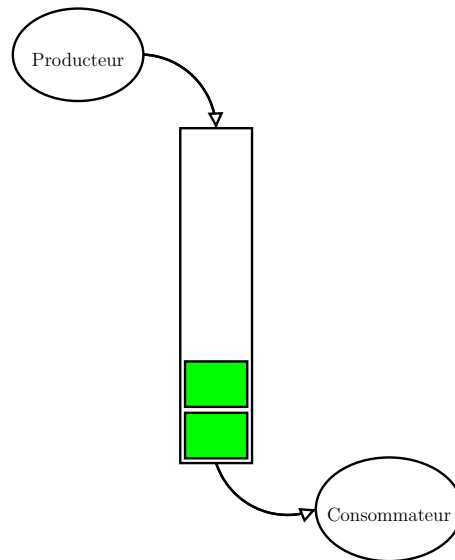


FIGURE 7.4 – Principe des queues de communication

Le producteur et le consommateur d'information :

- Ne doivent pas se bloquer l'un et l'autre
- Peuvent avoir des vitesses différentes (des temps de traitements différents)
- Peuvent s'exécuter à des moments différents

La queue de communication est un buffer permettant d'absorber les rythmes différents entre le producteur et le consommateur d'informations.

Les problèmes potentiels à gérer sont :

- Que faire quand c'est plein ? Comment gérer le fournisseur ? Doit-il se bloquer ou peut-on accepter de perdre des informations ?
- Que faire quand c'est vide ? Comment gérer le consommateur ?
- Est-ce résistant aux interruptions ?
- Est-ce résistant au multitâche ?
- ...

### 7.3.2 Compliquons un peu

Il n'y a pas toujours un seul producteur et un seul consommateur.

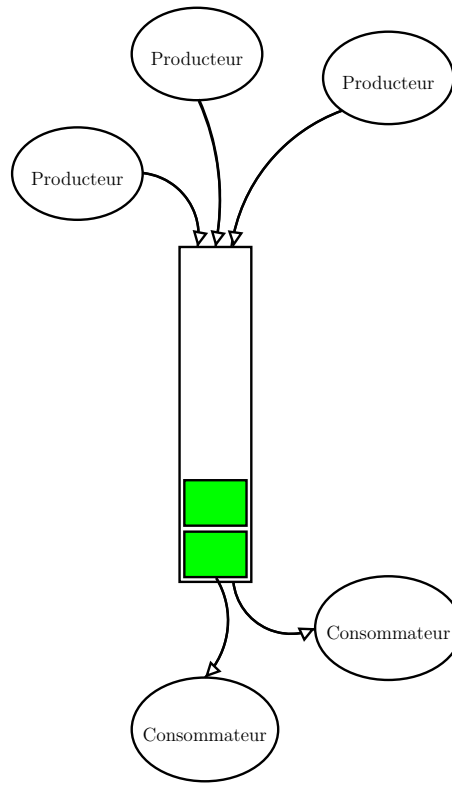


FIGURE 7.5 – Queue de communication, accès multiples

Au niveau des producteurs, il va y avoir :

- Un problème de concurrence, quelle tâche peut utiliser la queue de communication ?
- Un problème de remplissage quand une seule place est libre

Au niveau des consommateurs, les problèmes sont similaires :

- Concurrence ; quelle tâche doit prendre l'élément disponible ?
- Comment garantir qu'un élément est utilisé par une et une seule tâche en attente

Ces problèmes sont résolus pour nous par des fonctions de FreeRTOS.

De manière similaire aux autres fonctions de l'API de FreeRTOS :

- L'écriture ou la lecture dans une queue de communication peuvent être bloquants ou non (avec un timeout)
- Une seule tâche peut écrire dans la queue de communication (Mutex interne)
- Une seule tâche peut lire dans la queue de communication (Mutex interne)

- Si plusieurs tâches sont bloquées en lecture, la plus prioritaire (sinon la plus ancienne) sera débloquée en premier

Le listing 7.8 montre un exemple trivial contenant :

- Une tâche (**tacheProd**) qui génère des nombres aléatoires entiers et les transfère dans une queue de communication.
- Une tâche (**tacheConso**) qui consomme des entiers et en calcule la moyenne.
- Le programme principal (**main**) dont le rôle est de :
  - Créer une queue de communication destinée à stocker des entiers, d'une profondeur de 10 éléments.
  - Créer les 2 tâches (**tacheProd** et **tacheConso**).
  - Démarrer l'ordonnanceur.

Les priorités ne sont pas indiquées, mais au niveau du fonctionnement, si les priorités sont identiques :

- **tacheProd** va générer rapidement des nombres aléatoires et remplir la queue de communication. Elle va ensuite être bloquée, et ne se débloquent que quand la tâche **tacheConso** aura consommé un élément.
- La tâche **tacheConso** sera bloquée tant qu'il n'y a pas au moins un élément dans la queue de communication. Elle va ensuite l'utiliser pour ses calculs. Pendant ce temps, la tâche **tacheProd** peut continuer à produire des nombres.

On peut remarquer que les fonctions **xQueueReceive** et **xQueueSend** sont appelées avec le paramètre **portMAX\_Delay**. Il n'est donc pas nécessaire de tester le retour de ces fonctions, qui est obligatoirement **pdTRUE**.

```

1 QueueHandle_t q;
2
3 void tacheProd()
4 {
5     while(1)
6     {
7         int r = rand();
8         xQueueSend(q, &r, portMAX_DELAY);
9     }
10 }
11
12 void tacheConso()
13 {
14     long cumul = 0;
15     long count = 0;
16     double moy = 0.0;
17     while(1)
18     {
19         int v;
20         xQueueReceive(q, &v, portMAX_DELAY);
21         cumul += v;
22         count++;
23         moy = (double)cumul / (double)count;
24         // d'autre code éventuellement long...
25         // ...
26     }
27 }
28
29 void main()
30 {
31     q = xQueueCreate(10, sizeof(int));
32
33     xCreateTask(tacheProd, ...);
34     xCreateTask(tacheConso, ...);
35
36     vTaskStartScheduler();
37 }

```

Listing 7.8 – Utilisation de queue de communication

Dans une application réelle, on trouve en général des queues de communication :

- Avec plusieurs producteurs et un seul consommateur :
  - C'est le cas de **Gatekeeper**, utilisé pour protéger l'accès à une ressource critique.
- Avec un producteur et un consommateur :
  - Pour effectuer une gestion de données non bloquante pour le producteur.
  - C'est le cas des gestionnaires de commandes, on peut continuer à recevoir des commandes, tout en traitant les commandes précédentes.

- Les cas avec plusieurs consommateurs sont rares et peuvent correspondre à de la répartition de charge entre plusieurs microcontrôleurs ou FPGA externes.
  - A un niveau réseau d'entreprise, c'est ce qui se produit pour un gestionnaire de pool d'imprimantes, ou pour un répartiteur de requêtes web entre plusieurs serveurs.

### 7.3.3 Cycle de vie d'une queue de communication

La création d'une queue de communication se fait avec la commande :

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

- `uxQueueLength` est le nombre d'éléments qui seront stockés dans la queue de communication.
- `uxItemSize` est la taille de chaque élément.

Les queues de communication de **FreeRTOS** allouent initialement la **totalité** de la taille mémoire nécessaire au stockage de l'ensemble des éléments. Soit ici `uxQueueLength * uxItemSize` octets.

Contrairement à d'autres système d'exploitation temps réel, **FreeRTOS** effectue une copie des données dans la queue de communication. Il ne s'agit pas d'un simple transfert de pointeur.

Cela a l'avantage de permettre de modifier le buffer d'origine à partir du moment où il a été transféré dans **FreeRTOS** (avec une des commandes `xQueueSend...`).

Mais aussi l'inconvénient d'avoir potentiellement des temps de transferts importants si les éléments transférés ont des tailles importantes, et d'utiliser une grande place mémoire.

### 7.3.4 Transmettre des éléments d'une queue de communication

Le stockage d'un élément dans la queue de communication consiste à passer un pointeur sur l'élément à transférer, **FreeRTOS** connaît la taille des données à copier dans un des buffers de la queue de communication (tout est défini à la création).

```
BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvItemToQueue,
↳ TickType_t xTicksToWait);
```

Le paramètre `xTicksToWait` permet d'indiquer un temps maximum à attendre pour avoir une place de libre dans la queue de communication. Un

temps de 0 indique que l'on ne veut pas attendre si il n'y a pas de place disponible. Un temps défini à `portMAX_DELAY` indique que l'on souhaite attendre indéfiniment.

Ce paramètre va évidemment dépendre de l'application :

- Dans un flux audio, ou sur des données acquises au fil de l'eau, il peut être acceptable de ne pas transférer toutes les données (pour éviter les congestions de données similaires). On pourra alors mettre un timeout, ou même 0.
- Si les données sont critiques (pour un calcul), on pourra utiliser `portMAX_DELAY`. Aucune donnée ne sera perdue, mais la tâche sera bloquée lors du transfert si il n'y a pas de place disponible.

Il existe des fonctions de la famille `xQueueSend...` qui ont des utilisations particulières :

- `xQueueOverwrite` permet d'écrire dans la queue de communication même si elle est pleine. Cette fonction s'utilise avec des queues de communication ayant 1 seul élément. Le dernier élément (qui est aussi le premier) est alors remplacé.
- `xQueueSendToFront` permet d'écrire en tête de la queue de communication. Il faut quand même que de la place soit disponible. L'élément juste ajouté sera le prochain qui sera extrait par une des fonctions qui attend un élément.
- `xQueueSendToBack` est équivalent à la fonction de base `xQueueSend`.

Les fonctions `...FromISR` sont les seules qui peuvent être utilisées depuis une interruption.

### 7.3.5 Recevoir des éléments d'une queue de communication

La lecture d'un élément depuis une queue de communication consiste à passer un pointeur sur un élément à la fonction, qui remplira cet élément si il y en a un de disponible.

Cela n'est pas toujours le cas, et il est toujours possible d'indiquer un temps d'attente pendant lequel la tâche va être bloquée.

La fonction de base pour recevoir un élément de la queue de communication est :

```
BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer, TickType_t
↳ xTicksToWait);
```

Il existe des fonctions qui ont des utilisations particulières :

- `xQueuePeek` permet de lire un élément de la queue de communication sans retirer cet élément de la queue.

- `xQueueReset` permet d'effacer le contenu de la queue de communication.
- `uxQueueSpacesAvailable` permet de déterminer le nombre d'éléments non affectés dans la queue de communication (les espaces libres).

Les fonctions `...FromISR` sont les seules qui peuvent être utilisées depuis une interruption.

### 7.3.6 Comment transférer des types complexes

Les queues de communications sont faites pour traiter un certain type d'éléments défini par sa taille initiale.

Mais il peut être parfois utile d'utiliser des types complexes pouvant contenir différents contenus.

Le plus simple dans ce cas là est d'utiliser une combinaison de structures et d'unions.

Le listing 7.9 montre la déclaration d'une structure de donnée `mes_data_t` contenant un indicateur sur son contenu `contenu`, et une union contenant des variables stockées à la même adresse (pour réduire l'empreinte mémoire).

```

1  #define MAX_NOM 80
2
3  enum {CONTENU_NONDEFINI=0, CONTENU_ENTIER, CONTENU_NOM, CONTENU_COMPLEXE};
4
5  struct cmplx
6  {
7      double re;
8      double im;
9  };
10
11  typedef struct
12  {
13      unsigned contenu;
14      union
15      {
16          int valeur;
17          char nom[MAX_NOM];
18          struct cmplx datacomplexe;
19      } c;
20  } mes_datas_t;
21
22  QueueHandle_t xc;
23
24  void main(void)
25  {
26      //...
27      xc = xQueueCreate(5, sizeof(mes_datas_t));
28  }

```

Listing 7.9 – Structure de données pour queues de communication

Le listing 7.10 montre une utilisation de cette structure. La tâche `tacheConsommation` attend des éléments de type `mes_datas_t` et suivant le contenu va effectuer une action potentiellement différente.

La tâche `tacheFournisseurEntier` est un exemple d'une tâche fournissant des éléments de type entier et les envoyant dans la queue de communication.



```
1 void tacheConsommation(void *p)
2 {
3     while(1)
4     {
5         mes_datas_t md;
6         xQueueReceive(xc, &md, portMAX_DELAY);
7         switch(md.contenu)
8         {
9             case CONTENU_NOM:
10                // utilisation de md.c.nom
11                break;
12             case CONTENU_ENTIER:
13                // utilisation de md.c.valeur
14                break;
15             case CONTENU_COMPLEXE:
16                // utilisation de md.c.datacomplexe
17                break;
18         }
19     }
20 }
21
22 void tacheFournisseurEntier(void *)
23 {
24     while(1)
25     {
26         vTaskDelay(pdMS_TO_TICKS(100));
27         mes_datas_t md;
28         md.contenu = CONTENU_ENTIER;
29         md.c.valeur = xTaskGetTickCount();
30         xQueueSend(xc, &md, pdMS_TO_TICKS(50));
31     }
32 }
```

Listing 7.10 – Utilisation de structures de données pour queues de communication

### 7.3.7 Comment transférer de gros volumes de données

Pour éviter de transférer de grosses structures de données (qui augmentent l’empreinte mémoire) et qui mettent plus de temps à être copiées, il est possible de transférer uniquement des pointeurs.

Les zones mémoires ne sont ainsi plus copiées, mais l’implémentation devra s’assurer qu’il ne peut pas y avoir de corruption de la zone mémoire d’échange.

## 7.4 De nouveaux concepts (v10)

La nouvelle version de **FreeRTOS** (v10) ajoute de nouvelles fonctionnalités. Ce sont des optimisations dues à une spécialisation des anciennes fonctions.

- Une méthode de communication rapide avec les tâches pouvant remplacer les sémaphores et les queues de communication.
- Un moyen de communication optimisé entre deux tâches.

### 7.4.1 Direct to task notification

#### 7.4.1.1 Principe

Le mécanisme de **DirectToTask notification** permet dans certains cas (voir en 7.4.1.2) de remplacer :

- Les queues de communication simple (avec un entier comme type d'élément, et 1 seule élément).
- Les sémaphores binaires.
- Les sémaphores compteurs.
- Les groupes d'évènements.

Le principe est d'utiliser une valeur atomique intégrée dans le descripteur de la tâche. Les accès n'ont pas besoin d'être protégés, et les fonctions basées sur les **DirectToTask notification** sont alors plus performantes. L'utilisation des **DirectToTask notification** est similaire aux appels des fonctions liées aux sémaphores.

#### 7.4.1.2 Restrictions d'utilisation

Il n'est possible d'utiliser les **DirectToTask notification** que quand il n'y a qu'un récepteur en attente (du sémaphore ou de l'évènement).

### 7.4.2 Stream Buffers et Message Buffers

#### 7.4.2.1 Principe

Les **StreamBuffer** et **MessageBuffer** sont similaires dans le fonctionnement à des queues de communication simplifiées.

Les **MessageBuffer** sont créés à partir des **StreamBuffer** et en surchargent les fonctions pour permettre des transferts de données de tailles variables.

#### **7.4.2.2 Restrictions d'utilisation**

Il ne peut y avoir qu'un seul émetteur, et un seul récepteur. Si il doit y en avoir plusieurs, ce sera au développeur de gérer les accès concurrents (avec des `MutEx` par exemple).



# 8

## Interruptions, timers

*Ce chapitre traite de la gestion des interruptions sur un système d'exploitation temps réel comme **FreeRTOS**, et du bon traitement sur un **STM32F401**.*

*“Never interrupt someone doing what you said couldn't be done.”*

- Amelia Earhart,

### 8.1 Gestion des interruptions

Des évènements vont se produire pendant le fonctionnement de l'application, ils vont changer le déroulement prévu de l'application. On va trouver des évènements :

- Synchrones
  - Branchement sur exception : (trap, fault, abort, ...)
    - Division par zéro
    - Dysfonctionnement
    - Zone mémoire non autorisée
    - ...
  - Ils se produisent après l'exécution d'une instruction
- Asynchrones
  - Interruptions, en général sur des causes matérielles :
    - Boutons poussoirs
    - Fin de conversion d'un CNA
    - Arrivée d'un caractère sur une liaison série
    - Demande d'action provenant d'un périphérique
    - Fin d'un timer
    - ...

- Elles peuvent se produire à tout moment (et donc potentiellement pendant l'exécution d'une instruction), sauf si les interruptions ont été bloquées.

### 8.1.1 Interruptions dans un système temps réel

Les interruptions permettent de réagir :

- sans polling de tous les événements possibles du système
- sans ajouter de hardware coûteux
- "à la demande" d'interruptions venant du hardware

Les registres du contrôleur d'interruptions peuvent être pilotés par les fonctions **CMSIS**. Ceci assure une plus grande portabilité du code, mais aussi une meilleure lisibilité.

Les fonctions sont :

Autoriser et interdire une interruption :

```
void NVIC_EnableIRQ(IRQn_Type IRQn)
void NVIC_DisableIRQ(IRQn_Type IRQn)
```

Lire et définir l'état courant d'une interruption :

```
void NVIC_SetPendingIRQ(IRQn_Type IRQn)
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)
```

Lire et définir la priorité d'une interruption :

```
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
uint32_t NVIC_GetPriority(IRQn_Type IRQn)
```

Définir la répartition priorités-sous priorités des interruption :

```
void NVIC_SetPriorityGrouping(uint32_t priority_grouping)
```

Lire le numéro d'interruption de l'interruption active :

```
uint32_t NVIC_GetActive (IRQn_t IRQn)
```

### 8.1.2 Gestionnaire d'interruption

Dans l'architecture **ARM**, il y a un gestionnaire d'interruption par source d'interruption.

Au niveau de l'environnement, le fichier `startup_stm32f401xx.s` contient un tableau des vecteurs d'interruptions.

Le listing 8.1 montre une partie de ce fichier. Par défaut les interruptions pointent vers un `Default_Handler` qui n'est qu'une boucle infinie. Cela permet en cas d'appel d'une mauvaise interruption de bloquer complètement le programme (jusqu'à un `Reset`).

Le listing 8.2 montre le gestionnaire d'interruption par défaut. Le debugger peut se retrouver dans cette interruption en cas d'appel d'une interruption non définie.

Quand un gestionnaire d'interruption sera défini par l'utilisateur dans son programme, c'est ce gestionnaire qui sera appelé. Il suffit d'avoir une fonction ayant la bonne syntaxe, et elle sera appelée quand l'interruption correspondante sera déclenchée.

```
void EXTI1_IRQHandler(void)
{
    // gestion de l'interruption EXTI1
}
```

```

1  g_pfnVectors:
2  .word  _estack
3  .word  Reset_Handler
4  .word  NMI_Handler
5  .word  HardFault_Handler
6  .word  MemManage_Handler
7  .word  BusFault_Handler
8  .word  UsageFault_Handler
9  .word  0
10 .word  0
11 .word  0
12 .word  0
13 .word  SVC_Handler
14 .word  DebugMon_Handler
15 .word  0
16 .word  PendSV_Handler
17 .word  SysTick_Handler
18
19 /* External Interrupts */
20 .word WWDG_IRQHandler /* Window WatchDog */
21 .word PVD_IRQHandler /* PVD through EXTI Line detection */
22 .word TAMP_STAMP_IRQHandler /* Tamper and TimeStamps through the EXTI line */
23 .word RTC_WKUP_IRQHandler /* RTC Wakeup through the EXTI line */
24 .word FLASH_IRQHandler /* FLASH */
25 .word RCC_IRQHandler /* RCC */
26 .word EXTI0_IRQHandler /* EXTI Line0 */
27 .word EXTI1_IRQHandler /* EXTI Line1 */
28 .word EXTI2_IRQHandler /* EXTI Line2 */
29 .word EXTI3_IRQHandler /* EXTI Line3 */
30 .word EXTI4_IRQHandler /* EXTI Line4 */
31 .word DMA1_Stream0_IRQHandler /* DMA1 Stream 0 */
32 .word DMA1_Stream1_IRQHandler /* DMA1 Stream 1 */
33 .word DMA1_Stream2_IRQHandler /* DMA1 Stream 2 */
34 .word DMA1_Stream3_IRQHandler /* DMA1 Stream 3 */
35 .word DMA1_Stream4_IRQHandler /* DMA1 Stream 4 */
36 ...

```

Listing 8.1 – Table des vecteurs d'interruptions

```

1  .section .text.Default_Handler,"ax",%progbits
2  Default_Handler:
3  Infinite_Loop:
4      b Infinite_Loop

```

Listing 8.2 – Gestionnaire d'interruption par défaut, boucle infinie



### 8.1.3 Temps de latence

Les interruptions permettent au programme de réagir rapidement, mais pas de manière immédiate. Le calcul (ou l'estimation) du temps de latence maximal permet de déterminer si le système correspondra au cahier des charges.

Le temps de latence, est le délai entre l'évènement physique et la première ligne de code utile pour la gestion de cet évènement.

Le temps de latence maximal sera atteint quand :

- Les interruptions sont temporairement arrêtées (l'ordonnanceur est arrêté), ou si une interruption de priorité supérieure est en cours.
- Il faut terminer l'instruction en cours (pas la tâche en cours).
- Il faut sauvegarder le contexte de la tâche en cours (14 cycles sur un STM32F401)
  - registres
  - adresse
  - flags
  - pointeur de pile
  - ...

⇒ Le programme interrompu ne doit se rendre compte de rien ... à part le temps d'exécution qui augmente.

### 8.1.4 La cascade d'interruption (nesting)

Cela revient à une interruption dans une interruption.

- La gestion dépend de la cible
- On peut éventuellement gérer des priorités d'interruptions
- Il y a un empilement des contextes

### 8.1.5 Bonnes pratiques pour la gestion des interruptions

- Faire le moins possible de code à l'intérieur (sinon on risque de rater les interruptions de priorités inférieures)
- Différer le travail à une autre tâche
- Utiliser des fonctions de synchronisation
  - Rappel : En FreeRTOS, les fonctions utilisables dans les interruptions se terminent par `...FromISR`

## 8.1.6 Priorités des interruptions

### 8.1.6.1 Du point de vue de l'ARM

Quelques règles simples permettent de comprendre et d'utiliser les priorités du microcontrôleur ARM.

Les cœurs ARM permettent d'utiliser au maximum 256 niveaux de priorités (8 bit). Mais, en général juste une partie de ces bits est réellement disponible.

Une définition du CMSIS permet de découvrir combien il y a de bits réellement exploitable.

La constante `__NVIC_PRIO_BITS` contient le nombre de bits utilisables, qui est de 4 bits pour le STM32F401.

Les 16 priorités utiles sont codées sur les 4 bits de poids forts.

Priorités	Info
0x00	La plus prioritaire
0xn0	priorité intermédiaire n = 1..E
0xF0	La moins prioritaire

TABLE 8.1 – Priorités ARM

Les cœur ARM permettent de séparer les bits de priorités en 2 espaces, les bits de priorités et les bits de sous-priorités. Cela se configure avec la commande `NVIC_PriorityGroupConfig()`.

Mais il est conseillé de ne pas utiliser de sous-priorités et d'allouer tous les (4) bits aux priorités.

Dans nos programmes, nous utiliserons systématiquement la commande :

```
NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );
```

pour définir un espace de 4 bits de priorité.

### 8.1.6.2 Du point de vue de FreeRTOS

FreeRTOS sépare l'espace de 16 priorités en 2 groupes :

- Les priorités qui pourront être masquées par les sections critiques de FreeRTOS
- Les priorités qui ne pourront pas être masquées par les sections critiques de FreeRTOS

La frontière entre ces 2 groupes est une constante de FreeRTOS :

```
#define configMAX_SYSCALL_INTERRUPT_PRIORITY
```

```
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY    0xf
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5
#define configKERNEL_INTERRUPT_PRIORITY          (
↪ configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
#define configMAX_SYSCALL_INTERRUPT_PRIORITY      (
↪ configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
```

Listing 8.3 – Valeur de configuration des priorités

qui se trouve dans le fichier de configuration `freeRTOSConfig.h`. Le listing 8.3 montre les définitions du `freeRTOSConfig.h`.

Certaines valeurs sont dans la plage 0..15 (quand il y a 4 bits). Ce sont les valeurs directes (qui sont normalement utilisés par les appels CMSIS). Les autres valeurs sont décalées pour être directement utilisées dans les registres ou les codes assembleurs.

- `configLIBRARY_LOWEST_INTERRUPT_PRIORITY = 0xf`
- `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY = 5`
- `configKERNEL_INTERRUPT_PRIORITY = 0xf0`
- `configMAX_SYSCALL_INTERRUPT_PRIORITY = 0x50`

```
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI15_10_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority =
↪ configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY + 1;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

NVIC_DisableIRQ(EXTI15_10_IRQn);
NVIC_Init(&NVIC_InitStructure);
```

Listing 8.4 – Configuration de la priorité d’une interruption



### Choix des priorités

- `configMAX_SYSCALL_INTERRUPT_PRIORITY` ne doit jamais être égal à 0.
- Toutes les interruptions qui doivent utiliser des fonctions de FreeRTOS doivent utiliser les fonctions se terminant en `FromISR`.
- Toutes les interruptions qui doivent utiliser des fonctions de FreeRTOS doivent avoir une priorité inférieure à celle définie par `configMAX_SYSCALL_INTERRUPT_PRIORITY`, c’est à dire **une valeur numérique égale ou plus grande**.
- La fonction de définition de la priorité `NVIC_SetPriority()` ; utilise une valeur directe (0..15) si nous disposons de 4 bits pour la définition de la priorité, ainsi que les structures de définitions, comme indiqué dans le listing 8.4.

#### 8.1.7 Acquittement

Suivant le type d’interruption, il sera peut être nécessaire de l’acquitter. Cela se passe en général en réécrivant l’état courant de l’interruption ou en écrivant 0 dans le bit d’état.

Par exemple, le registre relatif aux interruptions des lignes extérieures est `EXTI_PR`, qui est encapsulé par la fonction `EXTI_ClearITPendingBit` de la `StdPeriph`.

#### 8.1.8 Interruption différée

Les interruptions qui vont avoir le droit d’utiliser des fonctions FreeRTOS ont une priorité plus basse (donc supérieure numériquement) à la priorité utilisée pour le tick de l’ordonnanceur.

Une bonne pratique est ensuite de faire le minimum d'opération dans l'interruption.

La figure 8.1 montre une bonne gestion d'une interruption :

1. Une tâche (tâche Handler) "Orange" va être bloquée, en attente d'un sémaphore binaire.
2. Une tâche s'exécute "Verte"
3. Une interruption (ISR) "Bleue" est déclenchée
4. L'interruption signale qu'une interruption s'est produite
5. Si la tâche "Orange" est de priorité supérieure à la tâche "Verte", elle est débloquée.

Une implémentation se trouve en 7.2.1, une interruption signale et débloquent une tâche.

Ce type d'implémentation permet :

- De ne pas rater d'interruptions.
- De différer le traitement de l'interruption si une tâche prioritaire est en cours d'exécution.

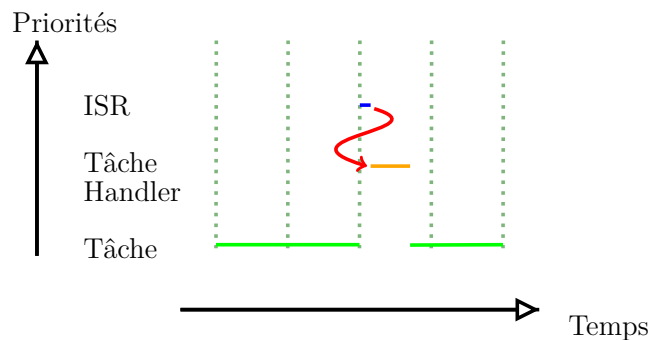


FIGURE 8.1 – Interruption différée

⇒ On utilise en général un sémaphore (binaire) de synchronisation, ou une queue de communication.

## 8.2 Timer

FreeRTOS permet de définir des timers logiciels (différents des timers hardware du microcontrôleur).

Les timers FreeRTOS sont gérés comme une tâche interne à FreeRTOS.

Il est ainsi possible d'exécuter des tâches périodiques en indiquant la fonction qui doit être appelée quand le timer expire (et si il faut recharger la valeur d'attente).

Je ne conseille pas l'utilisation de timers, car les fonctions ainsi appelées n'ont pas le droit d'être dans un état bloquant, comme un `vTaskDelay` par exemple. Ceci peut être contourné, mais le risque est important. Le développeur n'est pas averti s'il utilise une mauvaise fonction dans ce contexte et cela peut entraîner un arrêt du programme. Toutefois, dans la majorité des cas, les timers sont utilisés pour signaler un événement et n'ont pas de raisons de se bloquer en attente.

### 8.2.1 Configuration

Les fonctions de gestion des timers **FreeRTOS** sont de la famille `xTimerXXX`.

Comme pour les autres fonctions de l'API, un timer doit être créé avant d'être utilisé.

```
TimerHandle_t xTimerCreate( const char *pcTimerName, const TickType_t
↪ xTimerPeriod, const UBaseType_t xAutoReload, void * const pvTimerID,
↪ TimerCallbackFunction_t pxCallbackFunction );
```

Une des fonctions utiles de la famille des `xTimerXXX` est `xTimerStart` (et son équivalent `xTimerStartFromISR`) qui permet d'implémenter facilement un timeout sur une opération. Il suffit de rappeler `xTimerStart` chaque fois qu'un événement se produit. Quand l'événement ne se produit plus, la fonction associée au timer est appelée.

Le prototype est :

```
BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

Nous verrons un exemple d'exercice avec des timers, car ils sont très pratiques pour avoir automatiquement des gestionnaires d'interruptions différées, ou gérer des timeout après une action.

# 9

## Trace, débogage et analyse

*Ce chapitre traite de l'analyse du fonctionnement de **FreeRTOS**, et des outils utilisés pour tracer le fonctionnement d'un système temps réel.*

*“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”*

- Brian Kernighan,

Un problème immédiat pour l'analyse d'un système embarqué est que :

- Il y a un grand nombre d'évènements à analyser.
- Il n'y a en général pas la place de les stocker sur la cible.
- Les évènements se produisent quasiment simultanément du fait des interruptions et des changements de contexte.
- ...

Il n'est pas possible d'utiliser les outils standards de débogage :

- Mettre un point d'arrêt dans le programme et progresser en pas à pas.
- Effectuer des traces de tous les évènements dans un terminal (qui est lent par rapport aux phénomènes observés).
- ...

L'idée est alors d'instrumenter le code pour récupérer depuis l'extérieur quelques informations, en limitant les effets de l'instrumentation sur le fonctionnement de l'application.

Ceci est nécessaire car nous pouvons parfois rencontrer des soucis liés à des synchronisations, et l'ajout de code supplémentaires peut complètement changer l'ordonnancement des tâches existantes.

## 9.1 Instrumentation manuelle

### 9.1.1 Traces triviales

Dans des cas simples et non critiques, il est possible de venir ajouter ses propres fonctions de traces dans les tâches de **FreeRTOS**.

Le listing 9.1 montre une simple tâche dont le seul rôle est d'envoyer un message (**Alive:0** et **Alive:1** sur la liaison série).

Le principe est simple pour du test rapide ou du prototypage, mais n'est pas utilisable dans une vraie application :

- Il faut pouvoir facilement enlever et remettre les traces :
  - La version **Release** ou **Production** doit être la plus optimale possible, sans trace de débogage.
  - Mais il faut pouvoir facilement les remettre pour aider à diagnostiquer des problèmes.
- L'empreinte mémoire du **printf** est importante, et peut être inutilisable suivant la cible.
- La fonction **printf** n'est pas protégée contre les accès mutuel.
- La fonction ici n'est pas parfaitement identifiée comme étant une fonction de débogage.

On pourra s'affranchir de ces soucis, en développant ses propres fonctions de traces et en les protégeant par des **MutEx**.

Les traces de type **printf** pourront être remplacées par des envois d'informations dans une queue de communication, où nous pouvons sortir des informations de codage sur des **GPIO** pour les traiter ensuite avec un analyseur logique...

Mais tout ceci est déjà intégré dans **FreeRTOS** et expliqué en 9.2.

```
1 void vTaskDebug(void *pParameters)
2 {
3     const portTickType DELAY = pdMS_TO_TICKS(3000);
4     uint32_t alive = 0;
5
6     for(;;)
7     {
8         printf("Alive:%u\n", (unsigned)alive);
9         alive = 1-alive;
10        vTaskDelay(DELAY);
11    }
```

Listing 9.1 – Tâche de vérification du fonctionnement du système



## 9.2 Instrumentation intégrée dans FreeRTOS

### 9.2.1 Mécanisme intégré de trace

Toutes les actions de bas niveau effectuées par FreeRTOS sont déjà instrumentées, comme :

- Un changement de contexte
- L'envoi d'un élément dans une queue de communication
- Le blocage d'une tâche en attente d'un `MutEx`
- ...

Des macros sont déjà intégrées dans l'API. Par défaut ces macros ne font rien, et ne consomment donc ni ressources processeur, ni ressources mémoire.

Mais il est possible de les définir à une de nos fonctions qui sera donc automatiquement appelée quand l'action correspondante sera réalisée.

FreeRTOS utilise la définition `configUSE_TRACE_FACILITY` du fichier `FreeRTOSConfig.h`. Nous pouvons utiliser cette définition pour charger ou non les fonctions de traces.

Le listing 9.2 montre comment ajouter simplement l'inclusion d'un fichier de définition de traces à la fin du fichier `FreeRTOSConfig.h`.

```
1  #ifndef FREERTOS_CONFIG_H
2  #define FREERTOS_CONFIG_H
3  #include <stm32f4xx.h>
4
5  #ifdef __cplusplus
6  extern "C" {
7  #endif
8  //...
9  //...
10 //...
11 #if (configUSE_TRACE_FACILITY == 1)
12     #include "../src/traceFacility.h"
13 #endif //configUSE_TRACE_FACILITY
14 #ifdef __cplusplus
15 }
16 #endif
17 #endif /* FREERTOS_CONFIG_H */
```

Listing 9.2 – Ajout des fonctions de traces à `FreeRTOSConfig.h`

Le fichier `traceFacility.h` pourra être défini comme dans le listing 9.3.

Les macros qui sont disponibles commencent toutes par `trace` et se terminent par le nom de la fonction qu'elles tracent, avec des variantes

(suivant que l'opération s'est bien déroulée ou non). Les tables 9.1 et 9.2 contiennent la liste des macros disponibles.

Certaines macros acceptent un ou plusieurs paramètres qui devront être éventuellement implémentés dans les fonctions associées.

```

1  #ifndef TRACEFACILITY_H_
2  #define TRACEFACILITY_H_
3
4  // La liste de toutes les traces se trouve dans la documentation de freeRTOS
5  // Ou dans ce manuel
6  #define traceTASK_SWITCHED_OUT() vInfoForSwitchOut()
7
8  #endif /* TRACEFACILITY_H_ */

```

Listing 9.3 – Fichier de trace

Le listing 9.4 montre une Implémentation de fonction de trace. Elle est donnée à titre d'exemple, ou pour de simples tests, car elle utilise une ressource non protégée (la liaison série).

Une meilleure utilisation de ces macros, est de sortir de l'information sur des GPIO et d'effectuer une analyse avec un analyseur logique ou un oscilloscope numérique.

Il est aussi possible d'utiliser des extensions de FreeRTOS (développées par [percepio.com](http://percepio.com) ou [segger](http://segger.com)). Ces extensions utilisent les macros de trace FreeRTOS pour diffuser de l'information (par USB, UDP, ...) vers une application externe qui stocke les événements et les affiche à l'utilisateur.

Ces outils tiers sont fournis dans le dossier FreeRTOS+trace (voir aussi en <https://percepio.com/gettingstarted/> ou chez Segger). Ils nécessitent normalement l'achat de licences d'utilisation.

Il est possible d'utiliser une application de ST STM-STUDIO-STM32. Cet outil permet de visualiser le contenu de variables depuis une application externe.

```

1  void vInfoForSwitchOut(void)
2  {
3      TaskStatus_t pxTaskStatus;
4      vTaskGetInfo( NULL, &pxTaskStatus, pdFALSE, eRunning);
5
6      printf("%u\n", (unsigned)pxTaskStatus.xTaskNumber);
7  }

```

Listing 9.4 – Implémentation d'une fonction de trace

Macro	Rôle et appel
<code>traceTASK_INCREMENT_TICK()</code>	Interruption tick système
<code>traceTASK_SWITCHED_OUT()</code>	Changement de contexte, sortie de tâche
<code>traceTASK_SWITCHED_IN()</code>	Changement de contexte, entrée de tâche
<code>traceBLOCKING_ON_QUEUE_RECEIVE()</code>	La tâche se bloque <b>Take</b> (queue, sémaphore, <b>MutEx</b> , ...)
<code>traceBLOCKING_ON_QUEUE_SEND()</code>	La tâche se bloque <b>Give</b> (queue, sémaphore, <b>MutEx</b> , ...)
<code>traceQUEUE_SEND()</code>	<b>Give</b> (queue, sémaphore, <b>MutEx</b> , ...)
<code>traceQUEUE_SEND_FAILED()</code>	<b>Give</b> (queue, sémaphore, <b>MutEx</b> , ...)
<code>traceQUEUE_RECEIVE()</code>	<b>Take</b> (queue, sémaphore, <b>MutEx</b> , ...)
<code>traceQUEUE_RECEIVE_FAILED()</code>	<b>Take</b> (queue, sémaphore, <b>MutEx</b> , ...)
<code>traceQUEUE_SEND_FROM_ISR()</code>	<code>xQueueSendFromISR()</code>
<code>traceQUEUE_SEND_FROM_ISR_FAILED()</code>	<code>xQueueSendFromISR()</code>
<code>traceQUEUE_RECEIVE_FROM_ISR()</code>	<code>xQueueReceiveFromISR()</code>
<code>traceQUEUE_RECEIVE_FROM_ISR_FAILED()</code>	<code>xQueueReceiveFromISR()</code>
<code>traceTASK_DELAY_UNTIL()</code>	<code>vTaskDelayUntil()</code>
<code>traceTASK_DELAY()</code>	<code>vTaskDelay()</code>

TABLE 9.1 – Macros de trace (communes)

Macro	Rôle et appel
traceMOVED_TASK_TO_READY_STATE()	La tâche devient Ready
traceGIVE_MUTEX_RECURSIVE()	xSemaphoreGiveRecursive()
traceGIVE_MUTEX_RECURSIVE_FAILED()	xSemaphoreGiveRecursive()
traceQUEUE_CREATE()	xQueueCreate()
traceQUEUE_CREATE_FAILED()	xQueueCreate()
traceCREATE_MUTEX()	xSemaphoreCreateMutex()
traceCREATE_MUTEX_FAILED()	xSemaphoreCreateMutex()
traceGIVE_MUTEX_RECURSIVE()	xSemaphoreGiveRecursive()
traceGIVE_MUTEX_RECURSIVE_FAILED()	xSemaphoreGiveRecursive
traceTAKE_MUTEX_RECURSIVE()	xQueueTakeMutexRecursive()
traceCREATE_COUNTING_SEMAPHORE()	xSemaphoreCreateCounting()
traceCREATE_COUNTING_SEMAPHORE_FAILED()	xSemaphoreCreateCounting()
traceQUEUE_PEEK()	xQueuePeek()
traceQUEUE_DELETE()	vQueueDelete()
traceTASK_CREATE()	xTaskCreate()
traceTASK_CREATE_FAILED()	xTaskCreate()
traceTASK_DELETE()	vTaskDelete()
traceTASK_PRIORITY_SET()	vTaskPrioritySet()
traceTASK_SUSPEND()	vTaskSuspend()
traceTASK_RESUME()	vTaskResume()
traceTASK_RESUME_FROM_ISR()	xTaskResumeFromISR()
traceTIMER_COMMAND_RECEIVED()	Service timer, avant la commande
traceTIMER_COMMAND_SEND()	xTimerReset(), xTimerStop()
traceTIMER_CREATE()	xTimerCreate()
traceTIMER_CREATE_FAILED()	xTimerCreate()
traceTIMER_EXPIRED()	Avant l'appel

TABLE 9.2 – Macros de trace (rares)

### 9.2.2 Hook sur les tâches

FreeRTOS permet de définir un paramètre particulier pour chaque tâche avec la fonction :

```
void vTaskSetApplicationTaskTag( TaskHandle_t xTask, TaskHookFunction_t
↪ pxTagValue );
```

Cette fonction (rarement utilisée) n'est disponible que si la définition suivante a été ajoutée dans le fichier FreeRTOSConfig.h.

```
#configUSE_APPLICATION_TASK_TAG 1
```

Si le paramètre `pxTagValue` d'une tâche défini avec `vTaskSetApplicationTaskTag` est un pointeur de fonction, alors il est possible d'appeler cette fonction avec la commande suivante :

```
BaseType_t xTaskCallApplicationTaskHook(TaskHandle_t xTask, void
↳ *pvParameters);
```

Ceci est particulièrement utile avec les macros de traces vue en 9.2.1.

Le listing 9.5 montre la déclaration d'une macro appelant automatiquement la fonction définie dans une tâche (par `vTaskSetApplicationTaskTag`) chaque fois que la tâche est activée.

```
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook(pxCurrentTCB, 0)
```

Listing 9.5 – Utilisation du `xTaskCallApplicationTaskHook`

## 9.3 Utilisation de STM-STUDIO-STM32

Il est possible d'utiliser cette application pour visualiser en temps réel le contenu de variables de l'application.

En associant les techniques précédentes et la possibilité offerte par la surveillance des variables, il est possible de :

- Visualiser l'état des tâches (0 inactive, 1 active, ou un autre codage)
- Visualiser les interruptions
- ...

Toutefois, la communication doit se faire grâce à une des méthodes prévues (basées sur le port de programmation) ; et il faut faire attention à choisir le fichier `.elf` correspondant à l'application en cours d'exécution. Le nombre de variables qu'il est possible d'analyser est aussi limité.

La figure 9.1 montre l'utilisation de **STM-STUDIO** pour visualiser le contenu de variables. Ici nous analysons des variables entières qui sont modifiées par des états du programme. Il est aussi possible d'effectuer des calculs sur des variables, de faire des tests, de détecter des valeurs min/max, ...

Nous utiliserons **STM-STUDIO** pour analyser les codes produits par certains des exemples.

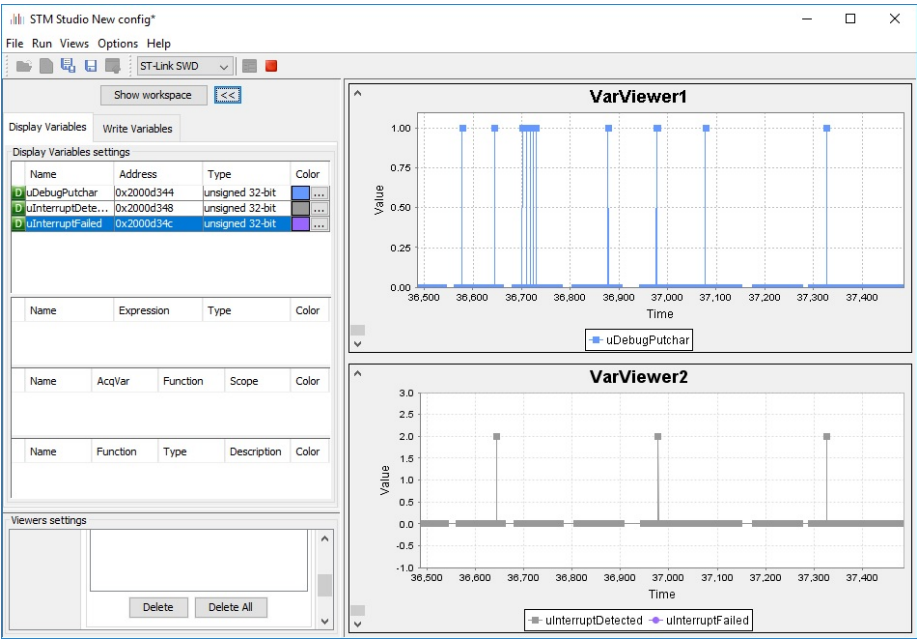


FIGURE 9.1 – STM Studio



## Annexe

### A.1 Configuration de FreeRTOS

Pour chaque nouvelle application, il est nécessaire de fournir un fichier `FreeRTOSConfig.h` contenant les réglages à appliquer à notre utilisation de `FreeRTOS`. Les dossiers d'exemples fournis avec `FreeRTOS` contiennent tous un fichier `FreeRTOSConfig.h`. Suivant les versions, ils peuvent contenir plus ou moins de définitions pré-remplies. On pourra se référer à la documentation en [6] pour la liste complète des entrées de ce fichier.

#### A.1.1 Les entrées commençant par `INCLUDE_`

Ces définitions permettent d'inclure ou d'exclure des parties des fonctionnalités de `FreeRTOS`. Cela permet de réduire l'empreinte mémoire utilisée, au détriment de la richesse fonctionnelle.

**Par exemple :** `#define INCLUDE_vTaskDelay 1` indique que la fonction `vTaskDelay` sera disponible. Sinon, elle ne sera pas compilée avec l'application.

#### A.1.2 Les entrées commençant par `config`

Ces définitions permettent de configurer le fonctionnement de `FreeRTOS`.

**Par exemple :** `#define configCHECK_FOR_STACK_OVERFLOW 1` indique que la stack doit être contrôlée. Dans ce cas là, la fonction de vérification devra être disponible. Elle permet en général de mettre un point d'arrêt

dans l'environnement de développement, et de pouvoir surveiller les variables internes.

## A.2 Définitions

**DMA** Direct Memory Access, le contrôleur de DMA peut être maître des bus pour effectuer des transferts entre la mémoire et les périphériques sans aucune action du processeur.

**NVIC** Nested Vectored Interrupt Controller est le gestionnaire d'interruption des microcontrôleurs à cœur ARM.

**octet** Un octet est un ensemble de 8 bits, il permet de stocker 256 valeurs différentes (de 0 à 255 en non signé, ou de -128 à +127 en signé).

**OTA** Over The Air, désigne les technologies permettant de distribuer et d'installer de nouvelles versions de firmware, des paramètres de configuration ou de nouvelles données dans un équipement mobile.

**portée** La portée d'une variable est la zone du programme ou son contenu est accessible. Une variable globale est accessible depuis l'ensemble du programme (éventuellement en indiquant qu'une variable est **extern** si elle n'est pas définie dans le même module). Une variable locale aura comme portée le bloc dans lequel elle se trouve.

**processus** Un programme binaire en cours d'exécution.

**programme** Une suite d'instruction à exécuter par un microprocesseur ou un microcontrôleur. On pourra distinguer les programmes sources (non encore compilés), et les programmes binaires (compilés et prêts à devenir un processus).



## A.3 Types de données et limites

Type	Min	Max
char	-128	127
unsigned char	0	255
short	-32 768	32 767
unsigned short	0	65 535
long	-2 147 843 648	2 147 843 647
unsigned long	0	4 294 967 295
long long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	0	18 446 744 073 709 551 615
float	$\pm 2^{-126} \approx \pm 1.175494 * 10^{-38}$	$\pm 2^{128} \approx \pm 3.402823 * 10^{38}$
double	$\pm 2^{-1022} \approx \pm 2.225074 * 10^{-308}$	$\pm 2^{1024} \approx \pm 1.797693 * 10^{308}$

TABLE A.1 – Plage de valeurs suivant le type

## A.4 Erreurs courantes

Les erreurs de syntaxes seront détectées par le compilateur (il est conseillé d'utiliser le mode `afficher tous les warnings`), mais certaines constructions seront valides syntaxiquement tout en étant (en général) fausses, c'est à dire que le résultat obtenu n'est pas celui attendu par le développeur.

Des outils de tests de code, permettent de détecter certaines erreurs.

- Division entière. `1/3` donne le résultat 0, il faut convertir au moins un des paramètres en flottants pour avoir une division flottante `1.0/3` ou `(double)(1)/3`.
- Utiliser une constante caractères `'a'` au lieu d'une constante chaîne de caractères `"a"` et inversement.
- Utiliser l'opérateur d'égalité `==` pour comparer des chaînes de caractères au lieu de la fonction `strcmp`.
- Ne pas penser au caractère de terminaison `\0` en fin de chaînes de caractères.

### A.4.1 Utiliser une constante caractères `'a'` au lieu d'une constante chaîne de caractères `"a"` et inversement

Erreur

```
char c;  
c = "a"; // erreur  
  
char *chaine = "bonjour";  
  
printf("Mon char:%c Ma chaine:%s\n",  
↪ c, chaine);
```

Valide

```
char c;  
c = 'a';  
  
char *chaine = "bonjour";  
  
printf("Mon char:%c Ma chaine:%s\n",  
↪ c, chaine);
```

### A.4.2 Utiliser = au lieu de == pour les comparaisons

Erreur

```
if (value = 1) // erreur, toujours  
↪ vrai
```

Valide

```
if (value == 1) // vrai test
```

### A.4.3 Accéder à un indice en dehors d'un tableau

Si un tableau est défini de la manière suivante :

```
int monTableau[10];
```

Erreur

```
monTableau[10] = 5; // erreur
```

Valide

```
monTableau[9] = 5; // ok, indice de 0  
↪ à 9
```

### A.4.4 Utiliser = ou ; dans un #define

Erreur

```
#define NB_INPUT = 5;
```

Valide

```
#define NB_INPUT 5
```

### A.4.5 Oublier d'initialiser une variable avant utilisation

Erreur

```
int nbElements;  
  
if (nbElements == 5) // erreur, non  
↪ initialisé
```

```
{  
    // Des actions  
}
```

Valide

```
int nbElements = 5;
```

```
if (nbElements == 5)
{
    // Des actions
}
```

#### A.4.6 Utiliser les opérateurs logiques (! || &&) au lieu des opérateurs bits à bits (~ & |)

Erreur

```
uint8_t a = !3; //a=0
uint8_t b = 3 || 5; //b=1
uint8_t c = 3 && 5; //c=1
```

Valide

```
uint8_t a = ~3; //a=0b11111100=252
uint8_t b = 3 | 5; //b=0b00000111=7
uint8_t c = 3 & 5; //c=0b00000001=1
```

#### A.4.7 Oublier un break dans un switch

Erreur

```
int a = 3;
switch(a)
{
    case 0:value=14;
    case 3:value=5;
    default:value=7;
}
// value = 7
```

Valide

```
int a = 3;
switch(a)
{
    case 0:value=14;break;
    case 3:value=5;break;
    default:value=7;break;
}
// value = 5
```

#### A.4.8 Affecter un tableau avec l'opérateur =

Erreur

```
int fibo[5] = {1,2,3,5,8};
int fibo2[5];

fibo2 = fibo; // erreur
```

Valide

```
int fibo[5] = {1,2,3,5,8};
int fibo2[5];

for(i=0;i<5;i++)
{
    fibo2[i] = fibo[i];
}
```

#### A.4.9 Effectuer une division entière au lieu d'une division à virgule flottante

1/3 donne le résultat 0, il faut convertir au moins un des paramètres en flottants pour avoir une division flottante 1.0/3 ou (double)(1)/3.

Erreur

```
float f = 1/3; // f = 0
```

Valide

```
float f = 1.0/3; // f = 0.333333
```

### A.4.10 Utiliser un pointeur non initialisé

Erreur

```
int *py;  
*py = 12; // crash
```

Valide

```
int x;  
int *py;  
py = &x;  
  
*py = 12; // x=12
```

## A.5 Mots-clefs du langage C

Les mots-clefs suivants sont ceux définis en ANSI C :

auto break case char const continue default do double else enum  
extern float for goto if int long register return short signed sizeof  
static struct switch typedef union unsigned void volatile while  
Suivant les compilateurs et environnements, d'autres mots-clefs peuvent  
exister : asm, inline, ...

## A.6 Aide-mémoire sur les registres GPIO

La liste complète des registres peut se trouver dans [8] (au chapitre 8.4).  
Les registres font tous 32 bits, mais pour certains seuls 16 bits sont utilisés.

**GPIO\_MODER** Ce registre contient 16 champs de 2 bits. Les bits 0 et 1  
définissent le mode de la broche 0, les bits 2 et 3 pour la broche 1, ...

Pour chaque broche :

00 entrée (input)

01 sortie (output)

10 fonction alternative

11 analogique

**GPIO\_OTYPER** Ce registre contient 16 champs de 1 bit.

Pour chaque broche :

- 0 sortie push-pull (push-pull output)
- 1 sortie à drain ouvert (open-drain output)

**GPIO\_OSPEEDR** Ce registre contient 16 champs de 2 bits.

Pour chaque broche :

- 00 low speed
- 01 medium speed
- 10 fast speed
- 11 high speed (les vitesses sont définies dans la documentation du composant)

**GPIO\_PUPDR** Ce registre contient 16 champs de 2 bits.

Pour chaque broche :

- 00 pas de pull-up, pas de pull-down
- 01 pull-up
- 10 pull-down
- 11 réservé

**GPIO\_IDR** Ce registre contient 16 champs de 1 bit.

Les 16 bits de poids faibles contiennent l'état de chaque broche.

**GPIO\_ODR** Ce registre contient 16 champs de 1 bit.

Une écriture dans ce registre permet de définir l'état de la broche correspondante, si elle est configurée en sortie.

**GPIO\_BSSR** Ce registre contient 2 masques de 16 bits.

Les bits  $b_{31}$  à  $b_{16}$  sont le **Reset mask**. Un bit mis à 1 dans ce masque, force le bit correspondant (numéro du bit - 16) à 0 dans le registre ODR.

Les bits  $b_{15}$  à  $b_0$  sont le **Set mask**. Un bit mis à 1 dans ce masque, force le bit correspondant à 1 dans le registre ODR.

En une seule écriture du **GPIO\_BSSR** il est possible de définir le contenu du registre **GPIO\_ODR**.

On trouvera souvent défini **BSSRH** et **BSSRL** qui permettent directement d'accéder aux parties haute et basse du registre.

**GPIO\_AFRL** et **GPIO\_AFRH** La configuration des **Alternate Functions** nécessitent 4 bits par broche. La configuration est répartie sur 2 registres. **GPIO\_AFRL** pour les broches 0 à 7, **GPIO\_AFRH** pour les broches 8 à 15.

Pour chaque broche :

0000 AF0	1000 AF8
0001 AF1	1001 AF9
0010 AF2	1010 AF10
0011 AF3	1011 AF11
0100 AF4	1000 AF12
0101 AF5	1001 AF13
0110 AF6	1010 AF14
0111 AF7	1011 AF15

## A.7 Aide-mémoire API FreeRTOS

### A.7.1 Queues de communication

Création d'une queue de communication :

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

Suppression d'une queue de communication :

```
void vQueueDelete( TaskHandle_t pxQueueToDelete );
```

Attente de données depuis une queue de communication :

```
BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer, TickType_t  
↪ xTicksToWait);  
BaseType_t xQueueReceiveFromISR( QueueHandle_t xQueue, void *pvBuffer,  
↪ BaseType_t *pxHigherPriorityTaskWoken );
```

Envoi de données dans la queue de communication :

```
BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvItemToQueue,  
↪ TickType_t xTicksToWait);  
BaseType_t xQueueSendToFront( QueueHandle_t xQueue, const void * pvItemToQueue,  
↪ TickType_t xTicksToWait );  
BaseType_t xQueueSendFromISR( QueueHandle_t xQueue, const void *pvItemToQueue,  
↪ BaseType_t *pxHigherPriorityTaskWoken );  
BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue, const void  
↪ *pvItemToQueue, BaseType_t *pxHigherPriorityTaskWoken );
```

Lecture du nombre d'éléments dans une queue de communication :

```
UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );  
UBaseType_t uxQueueMessagesWaitingFromISR( const QueueHandle_t xQueue );
```

Lecture du nombre d'espaces disponibles dans la queue de communication :

```
UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
```

Remplacement de l'élément de la queue de communication (nb element=1) :

```
BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void *pvItemToQueue );  
BaseType_t xQueueOverwriteFromISR( QueueHandle_t xQueue, const void  
↪ *pvItemToQueue, BaseType_t *pxHigherPriorityTaskWoken );
```

Récupération d'un élément sans le supprimer de la queue de communication :

```
BaseType_t xQueuePeek( QueueHandle_t xQueue, void *pvBuffer, TickType_t  
↪ xTicksToWait );  
BaseType_t xQueuePeekFromISR( QueueHandle_t xQueue, void *pvBuffer );
```

## A.7.2 Sémaphores et Mutex

Création d'un Mutex pour la protection de ressources critiques :

```
SemaphoreHandle_t xSemaphoreCreateMutex();
```

Création d'un Mutex récursif :

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex();
```

Création d'un sémaphore binaire :

```
SemaphoreHandle_t xSemaphoreCreateBinary();}
```

Création d'un sémaphore de comptage :

```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount, UBaseType_t  
↪ uxInitialCount );
```

Donner (ou rendre) un sémaphore ou un Mutex :

```
BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );  
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore, BaseType_t  
↪ pxHigherPriorityTaskWoken );
```

Prendre un sémaphore ou un Mutex :

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t  
↪ xTicksToWait );  
BaseType_t xSemaphoreTakeFromISR( SemaphoreHandle_t xSemaphore, signed  
↪ BaseType_t *pxHigherPriorityTaskWoken );
```

### A.7.3 Délais

Attente de `xTicksToDelay` ticks en temps relatif (depuis maintenant), la macro `pdMS_TO_TICKS()` peut être utilisée.

```
void vTaskDelay(TickType_t xTicksToDelay);
```

Attente de `xTicksToDelay` ticks en temps absolu (depuis la dernière activation), la macro `pdMS_TO_TICKS()` peut être utilisée.

```
void vTaskDelayUntil(TickType_t *pxPreviousWakeTime, TickType_t  
↪ xTimeIncrement);
```



## A.8 Table des caractères ASCII

b6 b5 b4 BITS b3 b2 b1 b0	0	0	0	0	1	1	1	1
	0	0	1	1	0	0	1	1
	0	1	0	1	0	1	0	1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL	16 DLE	32 SP	48 0	64 @	80 P	96 '	112 p
0 0 0 1	1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
0 0 1 0	2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
0 0 1 1	3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
0 1 0 0	4 EOT	20 DC4	36 \$	52 4	68 D	84 T	100 d	116 t
0 1 0 1	5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
0 1 1 0	6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
0 1 1 1	7 BEL	23 ETB	39 ,	55 7	71 G	87 W	103 g	119 w
1 0 0 0	8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
1 0 0 1	9 HT	25 EM	41 )	57 9	73 I	89 Y	105 i	121 y
1 0 1 0	10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
1 0 1 1	11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
1 1 0 0	12 FF	28 FS	44 ,	60 <	76 L	92 \ 	108 l	124 
1 1 0 1	13 CR	29 GS	45 —	61 =	77 M	93 ]	109 m	125 }
1 1 1 0	14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
1 1 1 1	15 SI	31 US	47 /	63 ?	79 O	95 ·	111 o	127 DEL





## Index

### B.1 Table des figures

1.1	Offre des STM32 . . . . .	8
1.2	Carte NUCLEO-F401RE . . . . .	9
1.3	Synoptique de la carte d'extension . . . . .	11
1.4	Boîtiers STM32F401RE . . . . .	13
1.5	Fonctions disponibles suivants les boîtiers (extrait) . . . . .	14
1.6	Pinout STM32F401RET . . . . .	15
1.7	Diagrammes des fonctionnalités . . . . .	16
1.8	Gamme des STM32 . . . . .	17
1.9	Instructions ARM-M suivant le cœur ARM . . . . .	18
1.10	Carte d'extension et carte NUCLEO . . . . .	20
1.11	Diagramme bloc du STM32F401 . . . . .	21
1.12	Mémoire et bus . . . . .	22
1.13	Organisation mémoire . . . . .	24
1.14	Périphériques et adresses mémoire . . . . .	25
1.15	Périphériques et adresses mémoire (suite) . . . . .	25
1.16	Périphériques et adresses mémoire (suite) . . . . .	26
1.17	Plan mémoire avec bit-banding . . . . .	27
1.18	Arbre des horloges . . . . .	32
1.19	Registre de contrôle des horloges . . . . .	33
1.20	I/O port . . . . .	35
2.1	Registres du STM32F401xD/xE . . . . .	50
2.2	Registres d'états du STM32F401xD/xE . . . . .	51
2.3	Traitement des instructions . . . . .	55
2.4	Architecture Harvard . . . . .	55

## LISTE DES PROGRAMMES

---

2.5	Architecture Von Neumann . . . . .	56
3.1	GPIO et USART . . . . .	64
3.2	Interface STM32 ST-LINK Utility . . . . .	68
4.1	Apollo Guidance Computer . . . . .	79
4.2	Des appareils modernes . . . . .	80
5.1	Cycle de vie des tâches . . . . .	100
7.1	Problème d'inversion de priorités sans héritage . . . . .	117
7.2	Mars pathfinder . . . . .	117
7.3	Problème d'inversion de priorités avec héritage . . . . .	118
7.4	Principe des queues de communication . . . . .	127
7.5	Queue de communication, accès multiples . . . . .	128
8.1	Interruption différée . . . . .	147
9.1	STM Studio . . . . .	156

## B.2 Liste des tableaux

2.1	Nom des quartets . . . . .	46
2.2	Représentation mémoire de quelques octets . . . . .	47
2.3	Big-Endian et Little-Endian . . . . .	53
8.1	Priorités ARM . . . . .	144
9.1	Macros de trace (communes) . . . . .	153
9.2	Macros de trace (rares) . . . . .	154
A.1	Plage de valeurs suivant le type . . . . .	159

## B.3 Liste des programmes

1.1	Macro pour le Bit-banding . . . . .	28
1.2	Activation de l'horloge pour des périphériques . . . . .	33
1.3	Utilisation des registres GPIO . . . . .	35
1.4	GPIO, led et bouton poussoir . . . . .	36
1.5	Utilisation du DMA . . . . .	38
1.6	Routine d'interruption du DMA . . . . .	39

1.7	Utilisation et configuration d'une interruption avec le <b>DMA</b> . . . .	40
1.8	Configuration du <b>SysTick</b> . . . . .	42
1.9	Gestion d'interruption du <b>SysTick</b> . . . . .	42
1.10	Configuration du <b>SysTick</b> . . . . .	42
1.11	Lecture de l'identifiant unique . . . . .	43
1.12	Lecture de la taille de la mémoire flash . . . . .	44
2.1	Exemple de calculs avec valeurs immédiates . . . . .	52
2.2	Exemple d'affectation avec des valeurs immédiates . . . . .	52
2.3	Exemple de programme en <b>C</b> . . . . .	57
2.4	Exemple de programme assemblé (extrait) . . . . .	58
3.1	Configuration avec <b>StdPeriph</b> . . . . .	65
3.2	Configuration sans <b>StdPeriph</b> . . . . .	66
3.3	Configuration avec la <b>HAL</b> . . . . .	66
3.4	Fichier d'en-tête pour des tests unitaires simples . . . . .	70
3.5	Tests unitaires simples, initialisation . . . . .	71
3.6	Tests unitaires simples, fonctions de tests . . . . .	72
3.7	Tests unitaires simples, rapport . . . . .	73
3.8	Fichier d'exemple pour des tests unitaires simples . . . . .	73
5.1	Adaptation à la fréquence de l'oscillateur externe . . . . .	90
5.2	Configuration de <b>FreeRTOS</b> . . . . .	91
5.3	Descripteur de tâche . . . . .	94
5.4	Listes internes de <b>FreeRTOS</b> . . . . .	95
5.5	Définition d'une liste <b>FreeRTOS</b> . . . . .	96
5.6	Définition d'un élément de liste <b>FreeRTOS</b> . . . . .	96
5.7	Une simple tâche . . . . .	101
5.8	Prototype du <b>xTaskCreate</b> . . . . .	101
5.9	Exemple de création de tâche avec <b>xTaskCreate</b> . . . . .	102
5.10	Prototype du <b>vTaskDelete</b> . . . . .	103
5.11	Structures de tâches . . . . .	104
5.12	Prototype du <b>vTaskPrioritySet</b> et <b>uxTaskPriorityGet</b> . . . .	105
6.1	Exemple de <b>vApplicationMallocFailedHook</b> . . . . .	109
7.1	Problème d'accès aux ressources . . . . .	112
7.2	Phénomène de Deadlock . . . . .	119
7.3	Structure pour un Gatekeeper . . . . .	120
7.4	Tâche Gatekeeper . . . . .	121
7.5	Envoi de données vers le Gatekeeper . . . . .	121
7.6	Tâche qui attend une interruption . . . . .	122

## LISTE DES PROGRAMMES

---

7.7	Utilisation de sémaphore de synchronisation depuis une interruption	123
7.8	Utilisation de queue de communication . . . . .	130
7.9	Structure de données pour queues de communication . . . . .	134
7.10	Utilisation de structures de données pour queues de communication	135
8.1	Table des vecteurs d'interruptions . . . . .	142
8.2	Gestionnaire d'interruption par défaut, boucle infinie . . . . .	142
8.3	Valeur de configuration des priorités . . . . .	145
8.4	Configuration de la priorité d'une interruption . . . . .	146
9.1	Tâche de vérification du fonctionnement du système . . . . .	150
9.2	Ajout des fonctions de traces à <code>FreeRTOSConfig.h</code> . . . . .	151
9.3	Fichier de trace . . . . .	152
9.4	Implémentation d'une fonction de trace . . . . .	152
9.5	Utilisation du <code>xTaskCallApplicationTaskHook</code> . . . . .	155

## Bibliographie

- [1] ARM. *ARM® and Thumb®-2 Instruction Set*, 2008. QRC0001\_UAL.pdf.
- [2] ARM. *Cortex™-M4 Devices Generic User Guide*, 2010. DUI0553A\_cortex\_m4\_dgug.pdf.
- [3] Richard Barry. *Mastering the FreeRTOS™ Real Time Kernel*, pre-release 161204 edition edition, 2016.
- [4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, second edition*. Prentice Hall, inc, 2nd edition, 1988.
- [5] Metraware. *Spécifications Carte Arduino IUT, Registres internes FPGA*. Metraware, 0004-0069-0001 rev d edition, 2016.
- [6] Amazon Web Services. *The FreeRTOS™ Reference Manual*, reference manual for freertos version 10.0.0 issue 1 edition, 2017.
- [7] ST. *Nucleo Schematics*, 2014. MB1136.pdf.
- [8] ST. *STM32F401xB/C and STM32F401xD/E advanced ARM®-based 32-bit MCUs Reference manual*, 2015. en.DM00096844.pdf.
- [9] ST. *STM32F401xD and STM32F401xE ARM® Cortex®-M4 32b MCU+FPU interfaces Datasheet*, 2015. en.DM00102166.pdf.
- [10] ST. *STM32F3, STM32F4 and STM32L4 Series Cortex®-M4 Programming manual*, 2016. en.DM00046982.pdf.
- [11] Don Wells. Extreme programming : A gentle introduction. <http://www.extremeprogramming.org/>, 2013. [En ligne ; Page disponible le 8-ctobre-2013].

- [12] Wikipédia. Multitâche — wikipédia, l'encyclopédie libre. <http://fr.wikipedia.org/w/index.php?title=Multit%C3%A2che&oldid=149267727>, 2018. [En ligne ; Page disponible le 5-juin-2018].
- [13] Wikipédia. Système embarqué — wikipédia, l'encyclopédie libre. [http://fr.wikipedia.org/w/index.php?title=Syst%C3%A8me\\_embarqu%C3%A9&oldid=147271985](http://fr.wikipedia.org/w/index.php?title=Syst%C3%A8me_embarqu%C3%A9&oldid=147271985), 2018. [En ligne ; Page disponible le 6-avril-2018].
- [14] Wikipédia. Temps réel — wikipédia, l'encyclopédie libre. [http://fr.wikipedia.org/w/index.php?title=Syst%C3%A8me\\_temps\\_r%C3%A9el&oldid=143884084](http://fr.wikipedia.org/w/index.php?title=Syst%C3%A8me_temps_r%C3%A9el&oldid=143884084), 2018. [En ligne ; Page disponible le 14-juin-2018].