

# Weather Station

Station Météo avec Datalogger

**Rapport d'Architecture Logicielle**

**Groupe G0 - Coordinateurs**

Microcontrôleur : PIC18F26K83 @ 64 MHz

Projet d'Électronique Appliquée

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Architecture en couches</b>	<b>2</b>
2.1	Description des couches . . . . .	2
<b>3</b>	<b>Organisation des fichiers</b>	<b>3</b>
<b>4</b>	<b>Point d'entrée et structure du main.c</b>	<b>3</b>
4.1	Structure du main.c . . . . .	3
4.2	Configuration bits . . . . .	3
<b>5</b>	<b>Séquence d'initialisation (app_main_init)</b>	<b>4</b>
5.1	Ordre d'initialisation . . . . .	4
5.2	Gestion des erreurs . . . . .	4
<b>6</b>	<b>Système d'interruptions (isr.c)</b>	<b>4</b>
6.1	Interruptions gérées . . . . .	4
6.2	Configuration du Timer0 (1 seconde) . . . . .	5
6.3	Configuration du Timer1 (10 millisecondes) . . . . .	5
6.4	Gestionnaire d'interruptions unifié . . . . .	5
6.5	Drapeaux globaux . . . . .	6
<b>7</b>	<b>Boucle principale (app_main_loop)</b>	<b>6</b>
7.1	Principe du scheduler non-bloquant . . . . .	6
7.1.1	Problème des délais bloquants . . . . .	6
7.1.2	Solution : scheduler non-bloquant . . . . .	6
7.1.3	Illustration temporelle . . . . .	7
7.2	Structure de la boucle . . . . .	7
7.3	Variables statiques . . . . .	7
7.4	Gestion de la période d'échantillonnage . . . . .	7
<b>8</b>	<b>Configuration matérielle (board.c)</b>	<b>7</b>
8.1	Mappage des broches . . . . .	8
8.2	Configuration PPS (Peripheral Pin Select) . . . . .	8
<b>9</b>	<b>Communication entre modules</b>	<b>8</b>
9.1	Flux de données . . . . .	9
9.2	Types de données partagés . . . . .	9
<b>10</b>	<b>Optimisations et cache RAM</b>	<b>9</b>
10.1	Principe du cache . . . . .	9
<b>11</b>	<b>Répartition du travail - Groupe G0</b>	<b>10</b>
11.1	Responsabilités du groupe coordinateur . . . . .	10
11.2	Points d'intégration avec les autres groupes . . . . .	10
<b>12</b>	<b>Synthèse</b>	<b>10</b>
12.1	Points clés de l'architecture . . . . .	10
12.2	Avantages de cette architecture . . . . .	10
12.3	Flux d'exécution complet . . . . .	11

# 1 Introduction

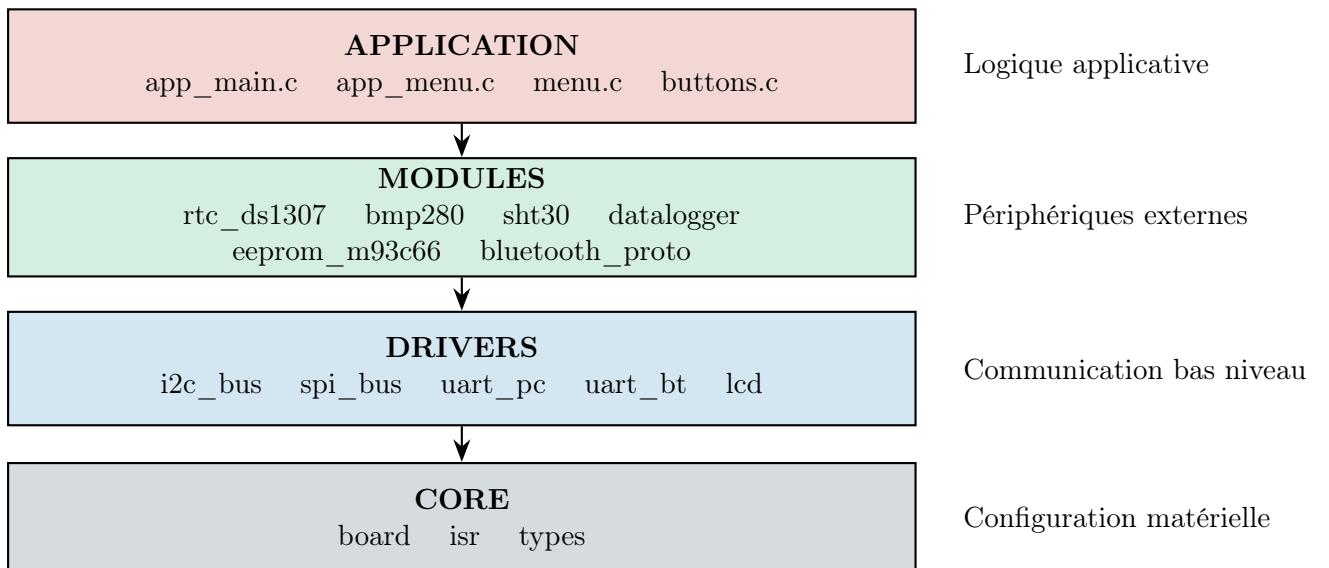
Ce document présente l'architecture logicielle complète de la station météo développée sur microcontrôleur PIC18F26K83. Il détaille particulièrement le rôle du groupe G0 (coordinateurs) qui est responsable de l'infrastructure de base et de l'orchestration du système.

Le projet consiste à développer une station capable de :

- Mesurer la température, l'humidité et la pression atmosphérique
- Afficher les données sur un écran LCD 16x2
- Enregistrer les mesures dans une mémoire EEPROM M93C66
- Transmettre les données via Bluetooth HC-05
- Gérer une interface utilisateur avec menu navigable

## 2 Architecture en couches

L'application est organisée en **quatre couches** distinctes, chacune ayant un rôle spécifique. Cette organisation permet de séparer les responsabilités et de faciliter le travail en équipe.



### 2.1 Description des couches

**CORE (G0)** : Cette couche configure le microcontrôleur lui-même. Elle définit quelles broches sont des entrées ou des sorties, comment les périphériques internes sont routés via PPS, et gère toutes les interruptions. C'est la fondation sur laquelle tout le reste repose.

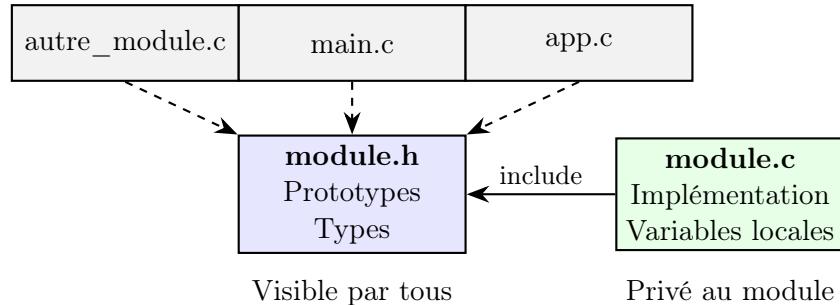
**DRIVERS (G0 + autres)** : Cette couche fournit des fonctions pour communiquer avec le monde extérieur. Elle encapsule les protocoles de communication (I2C, SPI, UART) et l'affichage LCD. Les modules au-dessus n'ont pas besoin de connaître les détails des registres du PIC.

**MODULES (tous les groupes)** : Cette couche contient la logique spécifique à chaque périphérique externe (capteurs, RTC, EEPROM, Bluetooth). Chaque module utilise les drivers pour communiquer avec son composant.

**APPLICATION (G0 + G1)** : Cette couche orchestre l'ensemble. Elle contient la boucle principale qui coordonne toutes les opérations, ainsi que l'interface utilisateur (menu LCD, gestion des boutons).

### 3 Organisation des fichiers

Chaque module est divisé en deux fichiers : un fichier d'en-tête (.h) et un fichier source (.c). Cette séparation est fondamentale pour le travail en équipe.



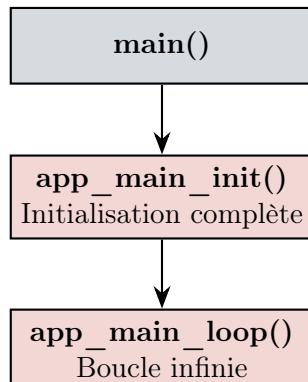
Le fichier .h contient les **prototypes** des fonctions (leur signature) et les **types** de données. Il est inclus par tous les autres modules qui ont besoin d'utiliser ces fonctions.

Le fichier .c contient l'**implémentation** réelle des fonctions. Son contenu est privé et peut être modifié sans affecter les autres modules, tant que les prototypes dans le .h restent identiques.

Cette organisation permet à chaque groupe de travailler sur son fichier .c sans créer de conflits avec les autres.

### 4 Point d'entrée et structure du main.c

Le programme démarre dans la fonction `main()` qui se trouve dans le fichier `main.c`. Cette fonction est volontairement très simple : elle appelle l'initialisation puis entre dans la boucle infinie.



#### 4.1 Structure du main.c

Le fichier `main.c` est le point d'entrée de l'application. Il contient la fonction `main()` qui appelle successivement l'initialisation complète du système puis entre dans la boucle infinie principale.

#### 4.2 Configuration bits

Les *configuration bits* sont des paramètres spéciaux du PIC qui définissent son comportement au démarrage. Ils sont programmés une seule fois et ne peuvent pas être modifiés durant l'exécution.

- **FEXTOSC = OFF** : Pas de quartz externe utilisé

- **RSTOSC = HFINTOSC\_64MHZ** : Oscillateur interne haute fréquence à 64 MHz
- **WDTE = OFF** : Watchdog timer désactivé (pas de reset automatique)
- **MCLRE = EXTMCLR** : Broche MCLR utilisée pour le reset externe

## 5 Séquence d'initialisation (app\_main\_init)

L'initialisation suit un ordre précis et critique. Chaque étape dépend de la précédente, et une erreur à n'importe quelle étape bloque l'exécution pour éviter un comportement imprévisible.

### 5.1 Ordre d'initialisation

1. Configuration matérielle (`board_init`)
2. Affichage LCD (`Lcd_Init`)
3. Bus I2C (`i2c_bus_init`)
4. Bus SPI (`spi_bus_init`)
5. UART1 et UART2 (`uart_pc_init, uart_bt_init`)
6. Périphériques I2C (`rtc, bmp280, sht30`)
7. EEPROM (`eeprom_init`)
8. Datalogger (`dl_reset_config`)
9. Bluetooth (`bluetooth_init`)
10. Menu (`app_init`)
11. Interruptions (`isr_init`)

### 5.2 Gestion des erreurs

Chaque étape retourne un code d'erreur de type `app_err_t`. Si une erreur survient, le système :

1. Affiche un message d'erreur sur le LCD
2. Entre dans une boucle infinie (`while(1);`)
3. Ne continue jamais l'exécution

Cette approche garantit qu'aucun module ne fonctionne sans que ses dépendances soient correctement initialisées. Par exemple, si le bus I2C échoue à l'initialisation, le système affiche "I2C Init Error" et se bloque.

## 6 Système d'interruptions (isr.c)

Le fichier `isr.c` centralise toutes les interruptions du système. Cette approche permet de gérer efficacement les événements asynchrones sans bloquer la boucle principale.

### 6.1 Interruptions gérées

1. **Timer0** : Tick périodique d'1 seconde pour le scheduler
2. **Timer1** : Tick de 10ms pour le debouncing des boutons et les animations du menu
3. **UART2 RX** : Réception de commandes Bluetooth
4. **IOC (Interrupt-on-Change)** : Détection des appuis sur les boutons

## 6.2 Configuration du Timer0 (1 seconde)

Le Timer0 génère une interruption toutes les secondes. Ce tick sert de base au scheduler de la boucle principale.

### Paramètres de configuration :

- Mode 16 bits
- Source d'horloge :  $F_{osc}/4 = 16 \text{ MHz}$
- Prescaler : 1 :256
- Fréquence résultante :  $16 \text{ MHz} / 256 = 62500 \text{ Hz}$

### Calcul de la précharge :

Pour obtenir 1 seconde, il faut 62500 cycles  
Précharge =  $65536 - 62500 = 3036 = 0x0BDC$

À chaque interruption, les registres TMR0H et TMR0L sont rechargés avec cette valeur pour maintenir la période d'1 seconde.

## 6.3 Configuration du Timer1 (10 millisecondes)

Le Timer1 génère une interruption toutes les 10ms. Il sert pour :

- Le debouncing des boutons (anti-rebond)
- Les animations du menu (clignotement, scroll)

### Paramètres de configuration :

- Mode 16 bits
- Source d'horloge :  $F_{osc}/4 = 16 \text{ MHz}$
- Prescaler : 1 :8
- Fréquence résultante :  $16 \text{ MHz} / 8 = 2 \text{ MHz}$

### Calcul de la précharge :

Pour obtenir 10ms, il faut 20000 cycles  
Précharge =  $65536 - 20000 = 45536 = 0xB1E0$

Cette interruption rapide permet une détection réactive des boutons et des animations fluides du menu.

## 6.4 Gestionnaire d'interruptions unifié

Toutes les interruptions passent par une fonction unique `__interrupt() isr_handler()`. Cette centralisation permet une gestion cohérente et facilite le débogage.

### Interruptions gérées :

1. **Timer0 (1s)** : Incrémente un drapeau pour le scheduler, recharge les registres TMR0H/TMR0L
2. **Timer1 (10ms)** : Incrémente un compteur pour les animations du menu, appelle la fonction de debouncing des boutons
3. **UART2 RX** : Stocke le caractère reçu et lève un drapeau pour traitement dans la boucle principale
4. **IOC (Interrupt-on-Change)** : Détecte les changements d'état des boutons et appelle la callback de gestion

Chaque interruption effectue le minimum de travail nécessaire et délègue le traitement complexe à la boucle principale via des drapeaux.

## 6.5 Drapeaux globaux

Les interruptions communiquent avec la boucle principale via des drapeaux globaux déclarés avec le mot-clé `volatile`. Ce qualificateur indique au compilateur que ces variables peuvent changer à tout moment (via interruption) et ne doivent pas être optimisées.

### Drapeaux utilisés :

- `g_timer0_flag` : Signale un tick de 1 seconde
  - `g_uart2_rx_flag` : Indique qu'un caractère Bluetooth a été reçu
  - `g_uart2_rx_char` : Contient le caractère reçu
  - `g_menu_tick_10ms` : Compteur incrémenté toutes les 10ms pour les animations
- La boucle principale vérifie périodiquement ces drapeaux et les remet à zéro après traitement.

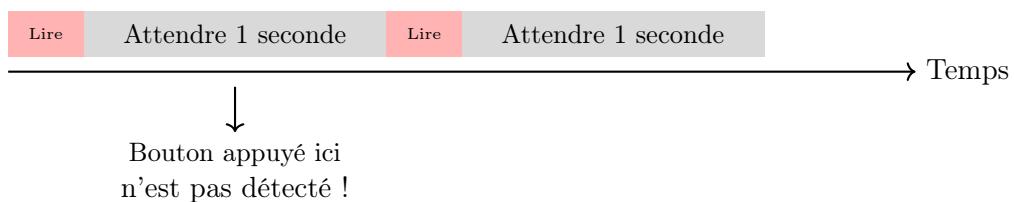
## 7 Boucle principale (`app_main_loop`)

La boucle principale orchestre toutes les tâches du système selon une architecture non-bloquante basée sur le scheduler.

### 7.1 Principe du scheduler non-bloquant

#### 7.1.1 Problème des délais bloquants

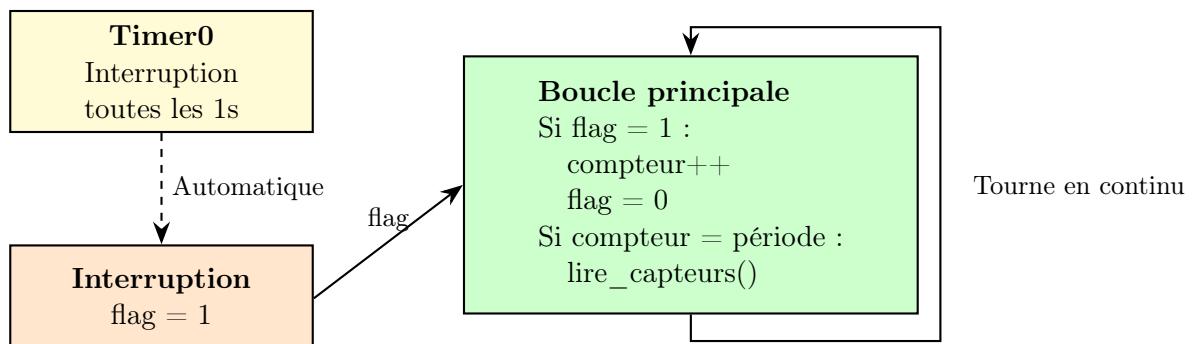
Une approche naïve consisterait à utiliser des délais pour espacer les lectures :



Pendant l'attente, le processeur ne fait rien et ne peut pas détecter les appuis sur les boutons. L'interface devient non-réactive.

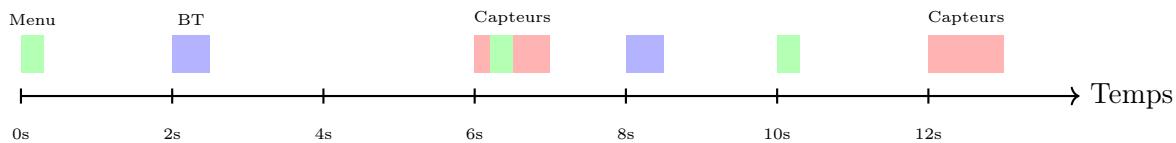
#### 7.1.2 Solution : scheduler non-bloquant

La solution utilise une interruption périodique (Timer0) qui lève un drapeau toutes les secondes. La boucle principale vérifie ce drapeau et compte les ticks.



Avec cette approche, la boucle principale tourne en permanence. Entre les lectures de capteurs, elle peut vérifier les boutons et mettre à jour l'affichage. Le système reste réactif.

### 7.1.3 Illustration temporelle



Le système ne bloque jamais. Entre deux lectures de capteurs, il traite continuellement le menu et les commandes Bluetooth.

## 7.2 Structure de la boucle

La boucle principale est organisée en plusieurs tâches qui s'exécutent selon des périodes différentes :

### Tâches continues (chaque itération) :

- Traitement des commandes Bluetooth reçues
- Mise à jour de l'interface utilisateur (menu)

### Tâches périodiques (basées sur Timer0) :

- Mise à jour de la configuration datalogger (toutes les 10 secondes)
- Acquisition des capteurs (période configurable, typiquement 1-5 minutes)

### Principe de fonctionnement :

La boucle utilise des variables statiques pour mémoriser l'état entre les itérations. Un compteur global (`g_tick_counter`) est incrémenté à chaque tick Timer0. Les tâches périodiques comparent ce compteur avec leur dernier moment d'exécution pour déterminer s'il est temps de s'exécuter.

Cette approche non-bloquante permet au système de rester réactif : pendant les longues périodes entre deux lectures de capteurs, le menu et le Bluetooth continuent de fonctionner normalement.

## 7.3 Variables statiques

Les variables `static` dans la fonction conservent leur valeur entre les appels. Cela permet de :

- Compter les ticks sans variable globale
- Mémoriser le dernier moment d'exécution de chaque tâche
- Éviter de polluer l'espace de noms global

## 7.4 Gestion de la période d'échantillonnage

La période d'échantillonnage est dynamique et configurée via le menu :

$$\text{Période en ticks} = \text{Période en minutes} \times 60 \text{ secondes}$$

Par exemple, pour une période de 5 minutes :

$$\text{Ticks} = 5 \times 60 = 300 \text{ secondes}$$

La configuration est relue toutes les 10 secondes pour détecter les changements faits par l'utilisateur dans le menu.

## 8 Configuration matérielle (board.c)

Le module `board.c` configure tout le matériel du PIC avant que les autres modules ne puissent l'utiliser.

## 8.1 Mappage des broches

Broche	Fonction	Pérophérique
<b>PORTA</b>		
RA0	RS	LCD
RA1	EN	LCD
RA2-RA5	D4-D7	LCD
RA6	BTN_UP	Bouton
RA7	BTN_DOWN	Bouton
<b>PORTB</b>		
RB1	SCL2	I2C (capteurs + RTC)
RB2	SDA2	I2C (capteurs + RTC)
RB3	CS	SPI (EEPROM)
RB4	LED1	Debug
RB5	LED2	Debug
RB6	TX2	UART Bluetooth
RB7	RX2	UART Bluetooth
<b>PORTC</b>		
RC0	BTN_ENTER	Bouton
RC1	BTN_BACK	Bouton
RC2	LED3	Debug
RC3	SCK	SPI (EEPROM)
RC4	SDI (MISO)	SPI (EEPROM)
RC5	SDO (MOSI)	SPI (EEPROM)
RC6	TX1	UART PC
RC7	RX1	UART PC

## 8.2 Configuration PPS (Peripheral Pin Select)

Le PIC18F26K83 utilise PPS pour router les périphériques internes vers les broches externes. Cela offre de la flexibilité mais nécessite une configuration explicite. Les codes hexadécimaux utilisés correspondent aux numéros de périphériques définis dans le datasheet du PIC.

**Exemple de routage :**

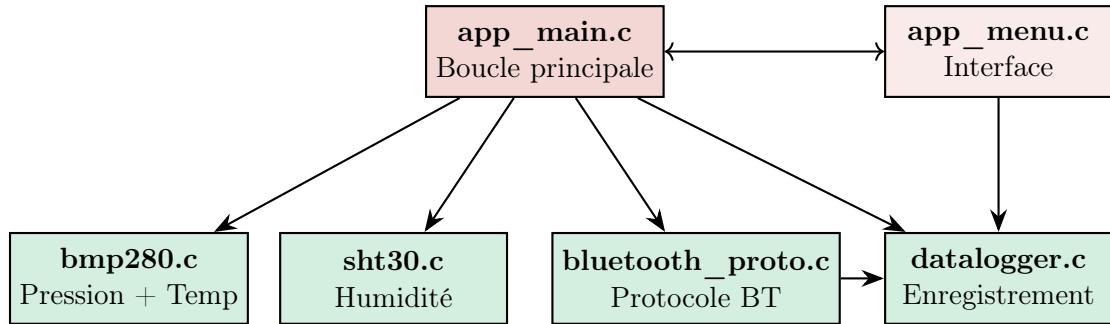
- **UART2 (Bluetooth)** : RB7 est configurée comme entrée U2RX, RB6 comme sortie U2TX
- **SPI1 (EEPROM)** : RC4 reçoit les données (SDI), RC5 émet (SDO), RC3 fournit l'horloge (SCK)
- **I2C2 (capteurs)** : RB1 et RB2 sont configurées en mode bidirectionnel pour SCL2 et SDA2

La configuration PPS est critique : une erreur de routage empêchera la communication avec les périphériques externes.

## 9 Communication entre modules

Les modules communiquent via des interfaces bien définies dans les fichiers d'en-tête.

## 9.1 Flux de données



## 9.2 Types de données partagés

Les structures de données sont définies dans `types.h` et partagées entre tous les modules. Cette centralisation garantit la cohérence des types utilisés.

**Structure des données capteurs :**

- `t_c_x100` : Température en degrés Celsius  $\times 100$  (ex : 2150 = 21.50°C)
- `rh_x100` : Humidité relative en pourcentage  $\times 100$  (ex : 6500 = 65.00%)
- `p_pa` : Pression atmosphérique en Pascals (ex : 101300 = 1013 hPa)

**Configuration du datalogger :**

- `sample_period_min` : Période d'échantillonnage en minutes
- `start_time` : Horodatage du démarrage (jour, mois, heure, minute)
- `data_count` : Nombre d'enregistrements actuellement stockés
- `running` : État du datalogger (1 = actif, 0 = arrêté)

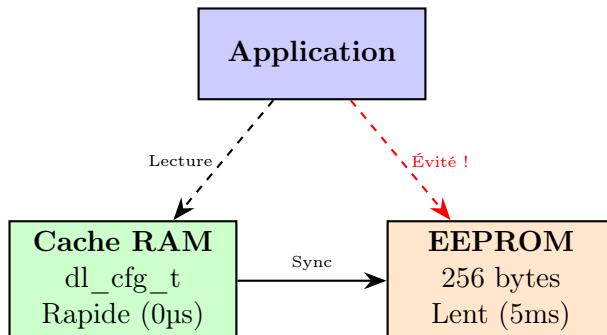
**Codes d'erreur unifiés :**

Tous les modules utilisent le même type énuméré `app_err_t` pour les codes de retour. Cela inclut `APP_OK` (succès), `APP_EPARAM` (paramètre invalide), `APP_EBUS` (erreur de communication), `APP_EDEV` (périphérique non présent), `APP_EFULL` (mémoire pleine), `APP_ENOTRUNNING` (datalogger arrêté), et `APP_ENOTCONFIG` (configuration manquante).

## 10 Optimisations et cache RAM

Le module `datalogger.c` utilise un système de cache pour minimiser les lectures EEPROM qui sont lentes (5ms par lecture).

### 10.1 Principe du cache



Le cache maintient une copie de la configuration en RAM. Les lectures sont instantanées et n'utilisent pas l'EEPROM.

## 11 Répartition du travail - Groupe G0

### 11.1 Responsabilités du groupe coordinateur

Le groupe G0 est responsable de l'infrastructure complète :

Module	Responsabilité
main.c	Point d'entrée, configuration bits
board.c/h	Configuration broches, TRIS, ANSEL, PPS
isr.c/h	Gestion centralisée des interruptions
types.h	Définition des types partagés
app_main.c/h	Boucle principale, scheduler, orchestration
i2c_bus.c/h	Driver I2C pour tous les périphériques I2C
spi_bus.c/h	Driver SPI pour l'EEPROM
uart_bt.c/h	Driver UART pour Bluetooth
uart_pc.c/h	Driver UART pour PC
lcd.c/h	Driver LCD 16x2 mode 4 bits

### 11.2 Points d'intégration avec les autres groupes

Groupe	Fichiers fournis	Interface utilisée
G1	menu.c, buttons.c, rtc_ds1307.c	app_init(), app_loop()
G2	bmp280.c	bmp280_init(), bmp280_read()
G3	sht30.c	sht30_init(), sht30_read()
G4	datalogger.c, eeprom_m93c66.c	dl_*, eeprom_*
G5	bluetooth_proto.c	bluetooth_handle_rx()

## 12 Synthèse

### 12.1 Points clés de l'architecture

- Initialisation séquentielle stricte** : Chaque couche dépend de la précédente, les erreurs bloquent l'exécution
- Scheduler non-bloquant** : Basé sur Timer0 (1s) et drapeaux, permet la réactivité
- Interruptions centralisées** : Un seul point d'entrée dans `isr_handler()`, gestion cohérente
- Séparation des responsabilités** : Chaque module a un rôle clair, pas de dépendances circulaires
- Communication via interfaces** : Les fichiers .h définissent les contrats entre modules
- Optimisation EEPROM** : Cache RAM pour minimiser les accès lents

### 12.2 Avantages de cette architecture

- **Travail parallèle** : Chaque groupe peut développer indépendamment
- **Testabilité** : Chaque module peut être testé isolément
- **Maintenabilité** : Les modifications sont localisées
- **Réutilisabilité** : Les drivers peuvent être réutilisés dans d'autres projets
- **Robustesse** : Gestion stricte des erreurs, pas de comportement indéfini

### 12.3 Flux d'exécution complet

