Pràctiques de Sistemes Digitals i Microprocessadors

Curs 2024-2025

Pràctica 2

Fase B

LSServerControl

Alumnes	Login	Nom
	alberto.marquillas	Alberto Marquillas Marsà
	alba.falcon	Alba Falcón Roy

Entrega	Placa	Memòria	Nota

Data	01/05/2025

Portada de la memòria

Índex

Resum de l'enunciat

La pràctica que es desenvolupa en aquesta segona fase del projecte gira entorn la creació d'un sistema embegut dissenyat per supervisar i gestionar les condicions tèrmiques d'una sala de servidors, un entorn especialment sensible a la temperatura a causa de la densitat de components electrònics i la seva contínua activitat. El microcontrolador utilitzat és el PIC18F4321, escollit per la seva versatilitat i pels recursos interns que ofereix, els quals, no obstant, són limitats i obliguen a un ús eficient de la memòria i del processador.

El sistema implementat ha de ser capaç de llegir la temperatura ambiental de manera periòdica a través d'un sensor analògic TMP36, i, en base als valors recollits, decidir l'estat del sistema: des d'un estat de temperatura baixa fins a un estat crític. Aquesta informació condiciona el comportament de dos ventiladors, que s'activen amb intensitats variables mitjançant senyals PWM, i d'un LED RGB que actua com a indicador visual de l'estat tèrmic del sistema. Cada estat (baixa, moderada, alta o crítica) implica una configuració específica dels ventiladors i del color del LED, sent l'estat crític especialment delicat: en aquest cas, per precaució, els ventiladors s'aturen completament per evitar propagar flames.

Complementàriament, el sistema ha d'enregistrar les temperatures mesurades en una memòria RAM externa (model 62256), que permet conservar una traça històrica del comportament tèrmic de la sala durant l'execució. A més, s'ha de mantenir un registre dels esdeveniments crítics, és a dir, quan es detecta una temperatura fora dels límits establerts, en la memòria EEPROM del propi microcontrolador. Aquests logs han d'incloure la data i hora exactes de l'esdeveniment, per la qual cosa s'utilitza un mòdul RTC (Real-Time Clock) de model DS3231, comunicat amb el microcontrolador mitjançant el protocol I2C. Aquest mòdul, a més, permet configurar alarmes per detectar quan ha transcorregut un minut exacte, funcionalitat que s'aprofita per enviar automàticament l'hora actual a la interfície gràfica.

Per tal de facilitar la interacció amb l'usuari, el sistema incorpora una connexió UART cap a un ordinador, des d'on s'executa una aplicació JavaFX que proporciona una interfície gràfica. Aquesta interfície actua com a panell de control del sistema i permet realitzar diferents accions: configurar els llindars de temperatura i el temps de mostreig, enviar la data i hora inicials al RTC, consultar els últims logs de temperatura crítica, obtenir totes les lectures enregistrades a la RAM per visualitzar-ne l'evolució mitjançant una gràfica, actualitzar l'hora del sistema sense reiniciar-lo i, finalment, reinicialitzar completament el sistema. L'usuari pot navegar per la interfície gràfica a través d'un joystick connectat físicament a la placa, utilitzant els eixos per desplaçar-se entre opcions i un polsador per confirmar seleccions.

Un dels requisits fonamentals de la pràctica és que tot el sistema s'ha d'implementar seguint una arquitectura cooperativa, evitant l'ús de tècniques bloquejants com delays o bucles d'espera activa. Cada mòdul del sistema (TAD) disposa d'un motor que s'executa cíclicament dins del bucle principal, cadascun d'aquests es troba dividit en fases que permeten avançar progressivament sense aturar l'execució global. Això garanteix una alta eficiència, especialment important en entorns amb recursos limitats com aquesta pràctica.

En resum, l'enunciat planteja el repte d'integrar múltiples perifèrics i funcionalitats dins d'un únic sistema embegut, amb un fort èmfasi en la modularitat, la cooperativitat i la gestió eficient de recursos. El resultat és un projecte transversal que posa a prova la capacitat d'anàlisi, disseny i implementació de sistemes digitals complexos mitjançant tècniques avançades de programació i control.

Plantejament del software

El sistema s'ha desenvolupat sota una arquitectura modular basada en TADs (Tipus Abstractes de Dades), cadascun dels quals s'encarrega d'una funcionalitat concreta del sistema. Aquesta estructura facilita tant el desenvolupament com el manteniment del codi, ja que permet separar clarament els diferents blocs de funcionalitat i afavoreix la reutilització i el depurat independent de cada mòdul.

El fil conductor del programa principal es fonamenta en un bucle infinit que actua com a planificador cooperatiu. A diferència d'un sistema amb multitasking preemptiu, en aquest cas cada mòdul implementa un motor (funció motor) que avança segons l'estat intern del mòdul i només ocupa temps de CPU durant la seva fase activa. Això permet executar múltiples tasques aparentment en paral·lel, garantint la fluïdesa del sistema i evitant bloquejos que podrien afectar el comportament global.

Quan el sistema arrenca, queda a l'espera d'una configuració inicial enviada per la interfície gràfica mitjançant UART. Aquesta configuració inclou la data i hora, els llindars de temperatura i el temps de mostreig. A partir d'aquest moment, s'activa el cicle de funcionament continu, que implica la lectura periòdica de la temperatura, l'activació proporcional dels ventiladors mitjançant PWM, l'actualització del LED RGB segons l'estat tèrmic i la gestió de l'RTC per detectar i notificar canvis horaris.

Les dades recollides es processen i es registren: les temperatures es desen a la RAM externa seguint una estratègia circular per optimitzar l'ús de memòria, mentre que els esdeveniments crítics es guarden a la EEPROM interna del microcontrolador, preservant-se fins i tot en cas de pèrdua d'alimentació. Per tal de garantir la sincronització horària, es fa ús d'una alarma del RTC configurada per saltar cada minut, moment en què el sistema envia l'hora actual a la interfície gràfica i comprova si s'ha de fer alguna acció addicional.

Aquest plantejament general del software constitueix l'esquelet principal sobre el qual es construeix tot el sistema. A continuació, s'aprofundirà en el funcionament específic de cada TAD que composa el projecte, explicant les seves responsabilitats, interaccions i motor cooperatiu corresponent.

Descripció dels TADs implementats

En aquesta secció s'expliquen en detall els diferents TADs que conformen el sistema. Cada TAD ha estat dissenyat per encapsular una funcionalitat concreta del projecte, seguint una filosofia modular i cooperativa. A continuació es descriuen un a un, indicant-ne la responsabilitat, funcionament intern i interaccions amb la resta del sistema.

TAD_ADC

El TAD_Adc és l'encarregat de gestionar el mòdul ADC (Analog-to-Digital Converter) del microcontrolador PIC18F4321. Aquest perifèric és essencial en sistemes embeguts quan es necessita llegir senyals analògiques, com és el cas del sensor de temperatura TMP36 i els eixos del joystick utilitzats en aquesta pràctica. Les entrades analògiques corresponen als pins AN0, AN1 i AN2 del port A del microcontrolador.

El sistema utilitza un motor cooperatiu que s'executa de forma cíclica dins del bucle principal del programa. Aquest motor es basa en una màquina d'estats senzilla implementada amb una variable global anomenada **`estat`**, que pot prendre tres valors: **`SELECT_CHANNEL`**, **`INICIA_CONVERSIO`** i **`CONVERSIO`**. A cada cicle d'execució, el motor avança d'estat si es compleixen les condicions requerides, permetent així una operació totalment no bloquejant. Això és essencial per mantenir la fluïdesa del sistema cooperatiu.

Inicialment, el TAD selecciona quin canal llegir (sensor de temperatura o un dels dos eixos del joystick), mitjançant l'escriptura al registre **`AD0_CH`**. El cicle de lectura segueix una rotació circular entre els tres canals: **`CH_TEMP`**, **`CH_JOYY`** i **`CH_JOYX`**. Un cop seleccionat el canal, l'estat passa a **`INICIA_CONVERSIO`**, on s'inicia la conversió configurant el senyal **`AD0_DONE`** a 1. La conversió és automàticament gestionada pel perifèric ADC.

Quan el senyal **`AD0_DONE`** torna a 0, indicant que la conversió ha finalitzat, el motor entra a l'estat **`CONVERSIO`**. En aquest punt, es llegeix el valor digital convertit del registre **`ADRESH`**, ja que s'ha configurat l'ADC amb format de justificació a l'esquerra (**`AD2_FORMAT = 0`**), per la qual cosa només cal consultar el byte alt. Aquest valor es desa en una variable interna segons el canal seleccionat: **`valor_temp`** si s'estava llegint el sensor TMP36, **`valor_joyx`** o **`valor_joyy`** en cas de lectura dels eixos del joystick.

Llavors, el canal s'incrementa per passar al següent en la seqüència i l'estat torna a **`SELECT_CHANNEL`**, preparant-se per una nova iteració. Aquest cicle de lectura cíclic permet tenir lectures actualitzades de cada entrada analògica en intervals regulars, de forma sincronitzada amb la resta de motors del sistema.

El mòdul es configura amb la funció **`ADC_InitADC`**, que inicialitza el perifèric ADC: activa el mòdul (**`AD0_ON = 1`**), defineix el temps d'adquisició (a través dels bits **`AD2_ACQUISITIONx`**), el temps de conversió (**`AD2_CONVERSIONx`**), el format de dades i la referència de tensió. Aquestes configuracions estan optimitzades per treballar amb el sensor TMP36 i garantir estabilitat i precisió en les lectures.

El TAD_Adc és un mòdul totalment passiu. No pren decisions ni modifica l'estat del sistema; simplement proporciona dades de sensors que seran utilitzades per altres mòduls. En concret, **`TAD_Temp`** transforma el valor digital de **`valor_temp`** en graus Celsius, mentre que

TAD_Joystick interpreta els valors de **valor_joyx** i **valor_joyy** per gestionar la navegació dins la interfície gràfica.

Les funcions d'accés **ADC_getTemp**, **ADC_getJoyX** i **ADC_getJoyY** permeten obtenir les últimes lectures de forma segura i no bloquejant. Aquestes funcions poden ser cridades en qualsevol moment, ja que els valors interns es van actualitzant automàticament per part del motor.

En resum, **TAD_Adc** constitueix una interfície essencial entre el món analògic i digital, permetent al sistema embegut accedir de manera periòdica, eficient i ordenada a valors provinents dels sensors, sense comprometre el rendiment ni la sincronització global del sistema cooperatiu.

TAD_Controlador

El **TAD_Controlador** constitueix el centre de decisió del sistema i és l'encarregat d'aplicar la lògica de control tèrmic en funció de les dades recollides pels sensors i els paràmetres establerts per l'usuari. El seu objectiu principal és garantir un comportament òptim del sistema davant canvis de temperatura, tot regulant els actuadors i gestionant la memòria de registre. El motor d'aquest TAD segueix un funcionament clarament estructurat mitjançant una màquina d'estats, amb el suport de diverses variables internes que controlen tant el flux de dades com l'estat general del sistema.

Inicialment, el controlador roman en estat d'espera fins que la configuració ha estat rebuda per part del mòdul de menú, **TAD_Menu**. Aquesta configuració inclou els llindars de temperatura que delimiten les diferents categories d'estat tèrmic (baixa, moderada, alta i crítica). Quan el **TAD_Menu** detecta que s'han introduït nous paràmetres, el controlador canvia d'estat i comença el seu funcionament normal. La funció **CONTROLER_resetController()** s'encarrega de reinicialitzar l'estat intern del sistema, posant a zero els indicadors de cicle de RAM i EEPROM, reiniciant el comptador de memòria i deixant el controlador llest per tornar a començar el cicle de supervisió.

Un cop el sistema ha rebut configuració, entra en el que es pot considerar el cicle operatiu típic. Si el mòdul **TAD_Temp** ha detectat que hi ha una nova lectura disponible i els motors de RAM i EEPROM estan disponibles, el controlador avança cap a l'estat on analitza la temperatura actual. Aquesta temperatura es recupera directament amb **TEMP_getTemp()** i es compara amb els valors de llindar que s'han emmagatzemat prèviament. El resultat d'aquesta comparació determina quin és l'estat tèrmic del sistema en aquell moment, que pot ser baixa, moderada, alta o crítica.

Segons l'estat tèrmic obtingut, el controlador activa les accions corresponents. Si l'estat és baix, es configura el LED RGB en verd i només un ventilador funciona a velocitat moderada. Si l'estat és moderat, s'activen els dos ventiladors també a velocitat mitjana i el LED canvia a blau. Si la temperatura és alta, els ventiladors operen a màxima potència i el LED es mostra en vermell. En el cas crític, s'apaguen els ventiladors per seguretat i s'activa el mode de parpelleig del LED entre vermell i magenta. Aquestes accions es gestionen mitjançant les funcions **LED_setLedColor**, **LED_setBlinkMode** i **VENT_setVelocity**, respectivament. A més, si l'estat passa a crític i no s'ha registrat cap log en aquest cicle, el controlador canvia d'estat i es prepara per escriure un registre a la memòria EEPROM.

Una vegada s'ha completat la lectura, interpretació i actuació en funció de la temperatura, el **TAD_Controlador** avança cap a les fases de registre, que s'organitzen en dos blocs: l'escriptura a RAM i, en cas necessari, a EEPROM. El procés de registre comença amb el valor de temperatura que ha estat emmagatzemat temporalment i que ara es vol persistir a la memòria RAM externa. Aquesta operació es fa en diverses etapes dins de la màquina d'estats del motor cooperatiu, començant per **TEMP_RAM**, on el valor es transfereix a una variable intermitja, i seguint amb **TEMP_RAM_INIT**, on s'escriu aquest valor a l'adreça actual de RAM. Tot seguit, en **TEMP_RAM_ESCRIVINT**, es comprova si l'adreça ha arribat al límit (32767), en quin cas es reinicia a zero per seguir un esquema circular. Finalment, en **TEMP_RAM_ESPERA**, s'activa el flag de disponibilitat de la RAM (**esticRam**) i el sistema passa a l'estat d'espera fins que torni a haver-hi una nova lectura disponible. Aquesta mecànica garanteix una traça de temperatura contínua i optimitzada dins del límit de memòria disponible.

Quan s'ha detectat un estat crític i encara no s'ha registrat en aquest cicle cap log a la EEPROM, el controlador entra a l'estat **TEMP_EEPROM**. Aquí s'utilitza la informació temporal prèviament llegida del mòdul RTC per construir un log de 14 bytes, en el format horari hhmmssDDMMYYYY. Aquest log es desa al buffer **cadenaEEPROM** mitjançant operacions directes sobre cada posició de la cadena, convertint els valors numèrics a caràcters ASCII. Un cop preparada, la cadena s'envia a **EEPROM_iniciaEscriptura**, sempre que no hi hagi cap operació en curs. El procés d'escriptura s'inicia i el motor avança a l'estat **ESPERA_ACABAR_EEPROM**, on s'hi manté fins que **EEPROM_HasAcabat() confirmi** que l'operació ha finalitzat. En aquell moment, s'activa de nou el flag **esticEEprom** i el controlador reprèn el cicle, passant a l'estat **TEMP_RAM** per completar el registre del cicle actual. Aquest sistema de control de registres a EEPROM utilitza també la variable **flagEE** per evitar que, dins un mateix episodi crític, es generin múltiples logs consecutius. Aquesta variable s'activa en el moment que es detecta la necessitat d'un log i es reinicia quan la temperatura retorna a un estat inferior. Aquesta estratègia millora l'eficiència de l'ús de memòria persistent i evita saturar el buffer amb dades redundants.

Aquest conjunt de comportaments defineix el funcionament estructurat del **TAD_Controlador**, que alterna entre la lectura, el processament, l'actuació i el registre de dades, coordinant-se amb altres mòduls del sistema i assegurant una gestió segura i eficient de l'estat tèrmic.

El **TAD_Controlador** ofereix també funcionalitats de consulta externa que permeten accedir a les dades guardades, especialment els logs crítics. La funció **CONTROLLER_getLogFormatted()** genera una cadena en format llegible que conté la data i hora d'un esdeveniment registrat a l'EEPROM. Aquesta funció utilitza un índex lògic que es transforma en índex físic real mitjançant **CONTROLLER_getLogIndex(),** el qual té en compte si la memòria ha completat una volta i si cal ajustar l'ordre de consulta per mantenir la cronologia dels esdeveniments. Aquesta informació pot ser enviada a la interfície gràfica mitjançant UART, i resulta útil per visualitzar l'historial d'alertes des de l'aplicació Java.

Finalment, el **TAD_Controlador** disposa de funcions per consultar i modificar l'adreça de RAM actual (**Controller_getRamAddr**, **Controller_clearRamAddr**) i per accedir als llindars configurats, així com al temps de mostreig. Aquest conjunt de funcionalitats complementàries permet una gestió flexible i remota del comportament del sistema, ja sigui per actualitzar paràmetres, restablir dades o consultar-ne l'estat. En conjunt, el **TAD_Controlador** manté un comportament central i coherent, coordinant el registre i l'actuació amb la resta del sistema i assegurant una supervisió tèrmica robusta i eficient.

TAD_Eeprom

El **TAD_Eeprom** s'encarrega de gestionar la memòria EEPROM interna del microcontrolador, amb l'objectiu de registrar i recuperar cadenes de 14 caràcters que representen esdeveniments crítics, com ara temperatures extremes. Aquestes cadenes es guarden seguint una estructura circular que permet conservar les 15 últimes entrades, i que es manté funcional fins i tot després d'un reinici del sistema gràcies al mecanisme de restauració d'estat.

En primer lloc, el mòdul ofereix una funció d'inicialització, **EEPROM_initEEprom**, que reinicia tots els estats interns del sistema de lectura, escriptura i reset. Aquesta inicialització assegura que tant els motors com les variables de control (adreces, comptadors, buffers, etc.) es troben en l'estat adequat abans de començar a operar.

El cor funcional del TAD es basa en l'escriptura cooperativa d'una cadena de caràcters, iniciada amb **EEPROM_iniciaEscriptura(const unsigned char* cadena)**. Aquesta funció configura el buffer d'escriptura i situa el motor en estat actiu. A partir d'aquest moment, **EEPROM_motorEscriptura()** s'encarrega de gestionar pas a pas l'escriptura de la cadena a la memòria. Cada caràcter s'escriu un a un, quan la memòria indica que està disponible (via **EEPROM_HasAcabat()**), assegurant que no es produeixen operacions bloquejants.

Quan s'ha escrit tota la cadena (14 caràcters + separador), el motor incrementa l'adreça d'escriptura per preparar la propera entrada. Si s'ha arribat al final de la memòria assignada per logs, l'adreça es reinicia a l'inici i s'activa el flag **full_turn**, indicant que la memòria circular s'ha completat almenys una vegada. En aquest moment, també s'incrementa el comptador intern de cadenes (**contador_cadenes**) fins a un màxim de 15.

A continuació, el mòdul escriu a les dues primeres posicions de la EEPROM: l'adreça d'escriptura actual (posició 0) i el nombre de cadenes registrades (posició 1). Aquest mecanisme permet, a l'inici de l'execució del sistema, recuperar l'estat previ mitjançant **EEPROM_restaurarEstat().** Aquesta funció valida que les dades emmagatzemades no siguin invàlides (per exemple, 0xFF o adreces fora de rang), i en cas contrari, reinicia els paràmetres corresponents.

Aquest mòdul inclou també funcions baix nivell per llegir (**EEPROM_Llegeix**) i escriure (**EEPROM_Escriu**) bytes individuals a la memòria. Aquestes funcions encapsulen la seqüència de passos requerida per accedir correctament a la EEPROM del microcontrolador, incloent el desbloqueig i la sincronització mitjançant registres especials com **EECON2** i **EECON1bits.WR**. Això permet una integració segura dins del model cooperatiu del sistema.

En conjunt, aquesta primera part del **TAD_Eeprom** permet una escriptura ordenada, cooperativa i persistent de logs de temperatura, assegurant que el sistema pugui recuperar en tot moment les condicions crítiques detectades, fins i tot després d'un reinici.

La segona part del **TAD_Eeprom** es centra en la recuperació de les dades registrades, el control de l'estat de lectura, la comprovació del torn complet, i la capacitat d'esborrar la memòria de forma cooperativa. Totes aquestes funcionalitats són essencials per garantir que la memòria EEPROM es pugui consultar i gestionar de manera eficient i segura dins el model cooperatiu del sistema.

Per a la lectura de cadenes, el TAD proporciona la funció **`EEPROM_iniciaLectura(unsigned char index, char* dest)`**, que inicialitza el procés de lectura d'un registre concret i defineix el buffer de destinació on es vol emmagatzemar la cadena. Aquesta lectura és controlada pel motor **`EEPROM_motorLectura()`**, que opera en dues fases: en primer lloc, calcula l'adreça base a partir de l'índex lògic de la cadena i de la mida fixa de les entrades; tot seguit, llegeix caràcter per caràcter la cadena seleccionada fins a completar-la, i escriu el caràcter nul **`\0`** al final per indicar el final de cadena. Durant aquest procés es fa ús d'un flag (**`lectura_completa`**) que permet consultar si la lectura ha finalitzat a través de la funció **`EEPROM_lecturaCompleta()`**.

La funció **`EEPROM_getContadorCadenes()`** retorna el nombre total de cadenes registrades, mentre que **`EEPROM_getFullTurn()`** indica si la memòria ha completat almenys una volta completa. Això és important per determinar si s'està treballant amb la memòria plena i si cal tenir en compte l'ordre cronològic invers en les lectures. L'adreça d'escriptura actual també pot ser consultada mitjançant **`EEPROM_getUltimaAddr()`**.

Per comprovar si s'està realitzant una escriptura en curs, es pot fer ús de **`EEPROM_esticEscriptura()`**, que retorna un valor cert si el motor d'escriptura es troba actiu. Aquesta funció és especialment útil per evitar conflictes entre processos cooperatius que intentin accedir simultàniament a la memòria.

Pel que fa al restabliment complet de la memòria EEPROM, el TAD proporciona dues opcions. D'una banda, la funció **`EEPROM_resetEEPROM()`** implementa un motor cooperatiu per esborrar de forma progressiva totes les adreces utilitzades (inclosos els dos primers bytes d'estat), escrivint-hi **`0xFF`** fins arribar al límit de memòria. Aquesta funció opera a través de diferents estats i manté un índex intern **`index_reset`** per saber fins on ha progressat l'esborrat. Quan finalitza, retorna un flag que indica que la memòria ha estat completament esborrada. D'altra banda, la funció **`resetEEPROMmain()`** permet esborrar de forma immediata tota la memòria en un sol bucle, però a diferència de l'anterior, no és cooperativa ni escalable per a ús dins el motor principal.

En conjunt, aquestes funcionalitats permeten gestionar de manera estructurada la consulta, control i reinicialització de la memòria EEPROM, garantint tant la persistència com la seguretat de les dades crítiques recollides pel sistema. Amb aquesta segona part, el **`TAD_Eeprom`** es completa com un mòdul robust, integrat plenament dins el model cooperatiu del projecte i amb suport total per a escriptura, lectura, consulta i esborrat.

TAD_JOYSTICK

El **`TAD_JOYSTICK`** és l'encarregat de gestionar la lectura del joystick connectat al sistema, el qual proporciona informació de moviment en dues dimensions, corresponents als eixos X i Y. Aquest joystick és un element essencial d'interacció en el sistema, ja que permet a l'usuari navegar per les diferents opcions de la interfície gràfica implementada en Java. Per fer-ho, el microcontrolador llegeix de forma periòdica els valors analògics dels dos eixos mitjançant el mòdul ADC, i els converteix en valors digitals que posteriorment s'interpreten per deduir la direcció del moviment.

Aquest TAD s'estructura funcionalment a partir de dues parts diferenciades: una funció de motor cooperatiu i una funció d'interpretació. La primera, anomenada **`JOY_MotorJoystick`**, s'encarrega de llegir els valors analògics actuals dels dos eixos del joystick i emmagatzemar-los en dues variables globals: **`valor_x`** i **`valor_y`**. Aquesta lectura es realitza mitjançant les funcions **`ADC_getJoyX()`** i **`ADC_getJoyY()`**, que encapsulen la lectura dels canals analògics assignats als eixos físics del dispositiu. El motor treballa de manera cíclica i no bloquejant, seguint l'estil cooperatiu requerit en la pràctica. Aquesta funció ha de ser cridada en cada iteració del bucle principal **`while(1)`** del programa, ja que no disposa de temporitzadors interns ni depèn d'interrupcions. Això assegura que els valors del joystick estiguin contínuament actualitzats i disponibles per a la resta del sistema.

Un cop obtinguts els valors, la funció **`JOY_getPosicio()`** s'encarrega de traduir aquesta informació numèrica en una posició simbòlica, és a dir, una de les constants: **`CENTRE`**, **`ADALT`**, **`ABAIX`**, **`DRETA`** o **`ESQUERRA`**. Aquesta decisió es basa en la comparació dels valors obtinguts amb dos llindars preestablerts: **`LLINDAR_INF`** i **`LLINDAR_SUP`**. Tot i que aquests valors poden variar segons la implementació, habitualment es troben dins un rang que garanteix la insensibilitat a petites variacions del senyal analògic, evitant així falsos posicions centrals. Si els valors dels dos eixos es troben dins del rang central delimitat per aquests llindars, el sistema interpreta que el joystick es manté en posició neutra. Si només l'eix X està centrat, s'analitza el valor de l'eix Y per deduir si el moviment ha estat a la dreta o a l'esquerra, i viceversa si només l'eix Y es manté centrat. Si cap dels dos eixos es troba en rang central, es retorna per defecte la posició **`CENTRE`**, un comportament que actua com a mesura de seguretat per evitar que moviments ambigus o senyals sorolloses desencadenin accions no desitjades.

Aquest TAD no actua de forma independent, sinó que col·labora amb altres mòduls del sistema, principalment amb el **`TAD_MENU`**, el qual consulta regularment la funció **`JOY_getPosicio()`** per determinar si l'usuari ha desplaçat el joystick i en quina direcció. Aquesta informació és essencial per navegar dins de la interfície d'usuari i seleccionar opcions, i representa una de les fonts d'entrada més crítiques durant l'execució normal del sistema. A més, altres TADs podrien fer-ne ús en un futur si s'estén la funcionalitat del sistema, com per exemple per controlar l'escaneig de logs o la navegació dins d'un historial de dades.

Una característica destacable d'aquest TAD és la seva simplicitat estructural. El motor consisteix en un únic estat (**`GET_VALORS`**) i no requereix gestió d'estats interns complexos ni temporitzacions específiques. Aquesta simplicitat és un avantatge dins del paradigma de sistemes cooperatius, ja que minimitza el temps d'execució dins de cada cicle i evita bloquejos o esperes innecessàries. No obstant això, la seva correcta funcionalitat depèn de la freqüència d'execució del bucle principal, ja que un retard excessiu podria provocar latència percebut en la resposta del sistema als moviments del joystick.

Finalment, cal remarcar que el disseny d'aquest TAD contribueix directament a la sensació de fluïdesa i control del sistema. Una detecció ràpida i fiable del moviment permet a l'usuari navegar pel menú amb precisió i confiança. Alhora, la resistència a petites desviacions gràcies a l'ús de llindars protegeix contra soroll elèctric o desplaçaments involuntaris. Així, el **`TAD_JOYSTICK`** no només compleix una funció tècnica, sinó que juga un paper clau en l'experiència d'usuari i en la qualitat percebuda del sistema implementat.

TAD_LED

El **TAD_LED** s'encarrega del control visual mitjançant un LED RGB que informa sobre l'estat tèrmic actual de la sala de servidors. Aquest LED esdevé un element clau d'indicació per l'usuari, ja que reflecteix de manera immediata i intuïtiva si la temperatura es troba dins de valors normals, elevats o crítics. El TAD es compon d'una inicialització, dues funcions setters, i un motor cooperatiu amb lògica condicional i temporal.

En primer lloc, la funció **LED_InitLed()** configura l'estat inicial del LED apagant totes les components de color (vermell, verd i blau) mitjançant crides a **OUT_ApagaLed()**, reinicialitza l'estat intern del TAD, i crea un temporitzador virtual amb **TI_NewTimer()** per controlar futurs parpelleigs. Aquesta configuració assegura que en el moment d'engegar el sistema no hi hagi cap color encès per defecte, i que l'estat inicial sigui sempre coherent i controlat.

El funcionament dinàmic del TAD està gestionat pel motor **LED_motorLed()**, una màquina d'estats senzilla que opera en tres fases: **MIRAR_BLINK**, **BLINK_MODE** i **MIRAR_COLOR**. Quan s'entra al motor, el primer estat comprova si el mode de parpelleig està activat a través de la variable **blink**. Si aquest està habilitat (estat crític), el motor canvia a l'estat **BLINK_MODE**, on alterna el color del LED entre vermell i magenta per indicar una situació d'emergència.

El parpelleig s'implementa mitjançant el temporitzador creat anteriorment. En concret, si el nombre de tics des del darrer canvi és inferior a 500, el LED es manté en color vermell (s'encén només el canal vermell); si és entre 500 i 1000, s'activa el canal blau a més del vermell, generant el color magenta. Quan els tics superen els 1000, es reinicia el comptador amb **TI_ResetTics()**. Aquesta oscil·lació visual alerta l'usuari de forma clara i repetitiva sobre una situació crítica. En tot moment, el canal verd es manté apagat per no interferir amb la percepció visual del parpelleig.

Quan no s'està en estat de parpelleig, el motor entra a **MIRAR_COLOR**, on actua segons el color que s'hagi configurat prèviament mitjançant la funció **LED_setLedColor()**. Aquesta funció setter emmagatzema el color que s'ha d'activar, i el motor s'encarrega d'encendre'l i apagar els altres dos canals per evitar colors mixtos involuntaris. Així, si el color actiu és vermell, s'apaguen els canals verd i blau, i s'encén exclusivament el vermell. El mateix es fa per qualsevol altre color triat, seguint una política d'exclusivitat.

A més, el TAD incorpora la funció **LED_setBlinkMode()** per activar o desactivar el mode de parpelleig des d'altres mòduls, com el **TAD_CONTROLADOR** o el **TAD_TEMPERATURA**, en funció de la gravetat del context tèrmic. Aquest enfoc modular permet que el LED actuï com a síntesi visual centralitzada de l'estat del sistema, sense que sigui necessari codificar la lògica visual dins de cada TAD funcional.

A nivell estructural, aquest TAD exemplifica una implementació cooperativa eficient basada en temporitzadors virtuals i estats ben delimitats. La separació clara entre el color estàtic i el mode dinàmic de parpelleig permet una fàcil extensió futura, com per exemple afegir nous codis de colors o integrar patrons addicionals.

Finalment, la presència d'aquest TAD en el sistema aporta una resposta immediata i no intrusiva a l'usuari, convertint el LED RGB en un canal de comunicació visual fiable i eficient. És una peça fonamental per a la supervisió passiva del sistema, especialment en situacions crítiques on no és viable consultar la interfície gràfica o esperar una resposta per UART.

TAD_MENU

El **TAD_MENU** és un dels components centrals del sistema, encarregat de gestionar tota la interacció entre la interfície Java i el microcontrolador. El seu objectiu és traduir les accions de l'usuari (rebudes a través del joystick o del polsador) en comandes que es puguin interpretar a la interfície gràfica, i a la vegada, executar les ordres que aquesta retorna al sistema. Tot aquest procés es gestiona mitjançant una màquina d'estats cooperativa implementada a la funció **MENU_MotorMenu()**.

Quan el sistema s'inicialitza, el menú entra en un estat de repòs (**ESTAT_JOY**) on resta a l'espera de moviment per part de l'usuari a través del joystick. Aquest moviment s'identifica mitjançant una crida a **JOY_getPosicio()**. En cas que es detecti un desplaçament, s'envia a la interfície Java un missatge ASCII que identifica la direcció (**UP**, **DOWN**, **LEFT**, **RIGHT**, **CENTER** o **SELECT**) i es passa a l'estat **ENVIANT_JOY**. Quan el moviment ha estat una selecció (determinada pel polsador), el sistema entra en **ESTAT_LLEGEIX**, on comença a llegir caràcter a caràcter la resposta enviada per la interfície. Aquesta es desa en un buffer fins detectar un salt de línia (**\n**), moment en què es transita a **ESTAT_AGAFA_DADES**.

En aquest nou estat es compara la cadena rebuda amb possibles comandes predefinides com **INITIALIZE**, **SET_TIME**, **GET_LOGS**, **GET_GRAPH** o **RESET**. En funció de la comanda detectada, es transita cap a un estat específic per gestionar la funcionalitat corresponent. En el cas d'**INITIALIZE**, es crida una funció que separa i desa els paràmetres rebuts (hora, data i llindars de temperatura), i es configura el RTC i el temps de mostreig. Per **SET_TIME**, es modifiquen les variables d'hora i minut i es passa a **ACTUALITZA_HORA**, on es consulta el RTC i es genera el missatge **UPDATETIME:HH:MM** que s'envia des de l'estat **ENVIA_HORA**.

Quan es demanen **GET_LOGS**, s'entra a un mecanisme que recorre els últims logs guardats a l'EEPROM i els envia línia per línia a la interfície Java, fins enviar el missatge final **FINISH**. Una lògica similar es segueix en **GET_GRAPH**, però en aquest cas es recorren les dades guardades a la RAM externa per generar una gràfica. Per a cada valor, s'envia una cadena **DATAGRAPH:XX** amb la temperatura llegida, fins acabar amb un **FINISH**. Tots aquests enviaments es fan de forma cooperativa i controlada per un temporitzador virtual per evitar col·lisions o repeticions innecessàries.

En cas d'executar un **RESET**, el sistema passa per diversos estats: **ESTAT_RESET_EEPROM**, **ESTAT_RESET_RAM** i **ESTAT_RESET_SYS**, els quals s'encarreguen respectivament d'esborrar la memòria EEPROM, la RAM, i reinicialitzar tots els TADs. Això inclou reconfigurar els perifèrics, reiniciar variables globals i preparar el sistema per una nova configuració.

Aquest TAD també incorpora una gestió interna d'alarma per detectar cada minut un canvi de temps proporcionat pel RTC. Quan salta aquesta alarma, el sistema entra en **ACTUALITZA_HORA** i s'encarrega d'enviar l'hora actualitzada a la interfície. Aquesta acció garanteix la sincronització contínua i autònoma entre el sistema embarcat i l'entorn gràfic.

En conjunt, el **TAD_MENU** articula la intel·ligència del sistema: és el nexe entre l'entrada de l'usuari, la resposta del sistema i la coordinació amb la resta de mòduls. La seva arquitectura basada en estats, temporitzadors i buffers permet una execució robusta, fluida i perfectament alineada amb el model de programació cooperativa adoptat en la pràctica.

A més del funcionament principal basat en una màquina d'estats, el **TAD_MENU** incorpora diverses funcions auxiliars i estructures internes que contribueixen al seu comportament

persistent i coherent al llarg del temps. Des del moment de la seva inicialització, a través de la funció **`initMenu()`**, es posa en marxa un temporitzador virtual anomenat **`TimerMenu`**, que té un paper essencial en el control de les esperes dins de certs estats i en la temporització d'enviaments, garantint així que l'execució del sistema es mantingui dins del model cooperatiu. Al mateix temps, es posa a zero el flag d'inicialització que indica si el sistema ja ha estat configurat correctament.

El TAD disposa d'una funció interna anomenada **`MENU_actualitzaEstat()`** que s'encarrega de detectar l'activació de dos tipus d'esdeveniments asíncrons: per una banda, una pulsació detectada pel polsador, que força el pas immediat al mode d'enviament de selecció; per l'altra, una alarma generada pel RTC que indica que ha transcorregut un minut. Si l'alarma s'activa i el sistema ja estava inicialitzat, s'activa l'estat de lectura d'hora per enviar-ne l'actualització a la interfície Java. Aquest comportament permet que el sistema es mantingui sincronitzat amb l'hora real, sense que sigui necessària cap acció per part de l'usuari.

Com a facilitador de la interacció amb altres mòduls del sistema, el **`TAD_MENU`** ofereix un conjunt de funcions que permeten consultar les variables internes que manté actualitzades. Aquestes funcions inclouen l'obtenció de la data i hora (**`MENU_GetHora`**, **`MENU_GetMinut`**, **`MENU_GetDia`**, **`MENU_GetMes`**, **`MENU_GetAny`**, **`MENU_GetSec`**), així com dels llindars de temperatura i del temps de mostreig configurats a l'inici del sistema. També es pot consultar si hi ha hagut una nova configuració des de l'última vegada, cosa especialment útil per a mòduls que necessiten ajustar-se a canvis rebuts des de la interfície Java.

Tot aquest sistema es recolza fortament en el temporitzador virtual inicialitzat al principi, que permet introduir retards no bloquejants i mantenir el ritme de resposta del sistema estable. Aquest temporitzador s'utilitza, per exemple, per introduir una pausa controlada després de l'enviament d'un moviment de joystick, evitant així relectures immediates o rebots, i també per temporitzar correctament la seqüència d'enviament de missatges per UART.

Pel que fa a la seva relació amb la resta del sistema, el **`TAD_MENU`** es comunica activament amb una gran quantitat de mòduls. Amb el **`TAD_JOYSTICK`** recull informació de moviment per a la navegació del menú. Amb el **`TAD_POLSADOR`** detecta confirmacions d'acció per part de l'usuari. Amb el **`TAD_SIO`**, gestiona l'enviament i la recepció de missatges amb la interfície Java. A través del **`TAD_RTC`**, pot llegir i configurar l'hora del sistema, ja sigui com a resposta a una comanda o com a manteniment periòdic. També accedeix al **`TAD_EEPROM`** per recuperar els últims logs del sistema i al **`TAD_RAM`** per generar les gràfiques amb les dades històriques de temperatura. Finalment, col·labora amb mòduls com el **`TAD_OUT`** i el **`TAD_CONTROLLER`** en accions com els reinicis complets del sistema o l'ajust dels paràmetres operatius interns.

Aquesta integració tan estreta i contínua fa que el **`TAD_MENU`** esdevingui el veritable centre de control del sistema, responsable tant de rebre accions de l'usuari com de dirigir el flux d'execució cap a la resposta adequada. La seva arquitectura basada en estats, funcions de consulta i temporitzadors cooperatius assegura una resposta fluida, modular i robusta que s'adapta amb precisió als requeriments de la pràctica i a les limitacions pròpies dels sistemes embeguts.

TAD_OUT

El **TAD_OUT** és un mòdul senzill però essencial dins l'arquitectura del sistema, responsable del control directe dels pins de sortida que alimenten el LED RGB. A diferència d'altres TADs amb motors o lògica d'estats, aquest actua únicament com una capa de baix nivell que permet encendre o apagar de manera individual cadascun dels tres canals de color del LED: vermell, verd i blau. Aquest enfocament modular facilita que altres TADs com ara el **TAD_LED** deleguin les accions físiques de control de maquinari a aquesta capa, mantenint la separació entre la lògica de decisió i la implementació física.

La funció **OUT_Init()** inicialitza l'estat dels tres canals del LED posant a zero els corresponents registres **LAT** associats als pins de sortida. Aquesta acció assegura que el sistema comença amb el LED apagat en totes les seves components, evitant efectes visuals no desitjats en el moment d'engegada. Aquesta inicialització és típica de sistemes embeguts on cal garantir un estat segur i conegut en el moment d'arrencada.

Les funcions **OUT_EncenLed()** i **OUT_ApagaLed()** reben com a paràmetre un codi identificador de color (**LED_COLOR_RED**, **LED_COLOR_GREEN** o **LED_COLOR_BLUE**) i actuen sobre el pin corresponent canviant el valor del registre **LAT** a 1 (per encendre) o a 0 (per apagar). Aquestes funcions encapsulen els detalls concrets del maquinari, permetent així que cap altre mòdul del sistema necessiti manipular directament els registres de sortida del microcontrolador. Això no només millora la llegibilitat i mantenibilitat del codi, sinó que també evita duplicació d'esforços o inconsistències entre mòduls que interactuen amb el LED.

Aquest TAD no disposa de cap motor ni estat intern, i el seu comportament és purament reactiu. És invocat exclusivament des d'altres TADs que gestionen el comportament lògic del LED, com ara el **TAD_LED**, que decideix quin color ha d'estar encès en funció de l'estat tèrmic del sistema o del mode de parpelleig actiu. Gràcies a aquesta divisió clara de responsabilitats, el **TAD_OUT** compleix un rol fonamental dins d'una arquitectura ben estructurada i coherent amb els principis de disseny modular.

TAD_POLSADOR

El **TAD_POLSADOR** és l'encarregat de gestionar de manera fiable la detecció d'una pulsació simple a través d'un botó connectat al microcontrolador. Aquesta funcionalitat resulta fonamental per capturar accions de selecció en el sistema, especialment durant la navegació per les opcions del menú. Atès que un polsador mecànic pot generar múltiples oscil·lacions elèctriques (rebots) en el moment de la pressió i de l'alliberament, aquest TAD implementa una màquina d'estats que inclou temporitzacions específiques per filtrar aquest comportament.

La funció **InitPolsador()** s'encarrega d'inicialitzar el sistema, configurant el pin associat com a entrada (**TRIS_POLS = 1**), posant l'estat inicial a **POLS_NO_PREMUT** i creant un temporitzador virtual que permetrà mesurar el temps transcorregut en cada fase de la detecció. També es posa a zero la variable **hiHaPolsacio**, que actuarà com a indicador intern de si s'ha produït una pulsació vàlida.

El cor del TAD és la funció **POLSADOR_MotorPolsador()**, que implementa una màquina d'estats amb quatre fases. En l'estat inicial, el sistema espera passivament que el pin de lectura passi a nivell alt, la qual cosa indica una possible polsació. Quan això succeeix, es passa a l'estat de **POLS_REBOTS_IN**, on s'espera un temps determinat (definit per la constant **REBOTS**) per deixar passar els rebots d'entrada. Si després d'aquest interval el polsador continua actiu, es confirma la pulsació i es passa a l'estat **POLS_PREMUT**.

A partir d'aquest moment, el sistema espera a que el polsador es deixi d'apretar. Un cop el senyal baixa, es torna a reiniciar el temporitzador i es passa a l'estat **POLS_REBOTS_OUT**, que filtra possibles rebots a l'alliberar el botó. Si passat el temps de seguretat el senyal continua baix, es confirma que la pulsació ha acabat correctament i s'activa el flag intern **hiHaPolsacio**.

Aquest flag pot ser consultat des de qualsevol altre TAD mitjançant la funció **POLSADOR_HiHaPolsacio()**, que retorna un 1 si hi ha hagut una pulsació recent, i 0 en cas contrari. Aquest mètode no reinicia el flag, ja que aquest queda a 0 automàticament a l'inici del següent cicle d'execució, dins l'estat **POLS_NO_PREMUT**.

El **TAD_POLSADOR** és un exemple clar d'un mòdul cooperatiu, lleuger i completament no bloquejant. Utilitza un temporitzador virtual per evitar retards actius i garanteix una detecció robusta fins i tot davant soroll elèctric o manipulació mecànica incerta. Aquesta implementació és especialment adequada per sistemes embeguts on el control del temps i la resposta precisa són imprescindibles per mantenir la fluïdesa del sistema global.

TAD_RAM

El **TAD_RAM** gestiona la comunicació amb una memòria externa estàtica del model 62256, utilitzada per emmagatzemar les lectures de temperatura realitzades pel sistema. Aquesta RAM s'organitza mitjançant un bus d'adreces de 15 bits i un bus de dades de 8 bits, amb les línies d'adreça A[8..14] controlades per un registre de desplaçament (74LS173) mitjançant un senyal de rellotge extern. El TAD encapsula tota la lògica necessària per escriure, llegir i reinicialitzar la memòria, de manera que altres mòduls del sistema poden operar sobre la RAM sense preocupar-se dels detalls de maquinari.

La funció **RAM_Init()** configura tots els pins implicats en la comunicació amb la memòria: les línies de control **WR**, **RD** i **CE** es defineixen com a sortides, el bus de dades s'inicialitza en mode lectura, i els busos d'adreça baixa i alta es preparen per la transmissió. Per defecte, s'activa el mode de lectura (**RD = 0**) per evitar conflictes al bus de dades i s'apaga l'escriptura (**WR = 1**). També es desactiva el rellotge que controla els registres d'adreces altes.

La funció **RAM_Write()** permet escriure un byte en una adreça concreta. Per fer-ho, en primer lloc es prepara la part alta de l'adreça, que s'envia al registre 74LS173 mitjançant el senyal de **RAM_CLK**, activant el rellotge per sincronitzar les línies A[8..14]. A continuació, s'especifica la part baixa de l'adreça directament a través del port B. Després es configura el bus de dades en mode escriptura i s'introdueix el byte a transmetre. Finalment, s'activa la línia **WR** per completar el cicle d'escriptura i es retorna el bus al mode lectura per seguretat.

El procés de lectura amb **RAM_Read()** segueix un patró molt similar. Es configuren les adreces alta i baixa com en l'escriptura, s'assegura que el bus està en mode entrada (**TRISD = 0xFF**) i es llegeix el valor del port D després d'una breu espera per estabilitzar les dades. Aquest enfoc garanteix una lectura robusta i segura, compatible amb les temporitzacions típiques del xip 62256.

Aquest TAD també incorpora una funcionalitat per reinicialitzar parcialment la memòria mitjançant la funció **RAM_resetRAM()**. Aquesta s'executa com una màquina d'estats cooperativa, recorrent seqüencialment totes les adreces escrites fins al moment (valor rebut com a paràmetre) i escrivint-hi zeros. Aquest procés es fa sense bloquejar el sistema i retorna un flag de finalització quan s'ha completat, cosa que permet al sistema reinicialitzar la RAM sense interferir amb altres motors actius.

La implementació del **TAD_RAM** parteix d'un disseny de baix nivell molt ajustat al maquinari. L'ús del registre 74LS173 per gestionar les adreces altes permet estalviar pins del microcontrolador, i el senyal **RAM_CLK** és gestionat manualment per assegurar que el registre rebi un flanc de pujada que faci efectiu el canvi d'adreça. Les instruccions **asm("NOP")** inserides entre operacions garanteixen que els senyals tinguin temps suficient per estabilitzar-se abans de procedir amb la lectura o l'escriptura, atenent així els requisits temporals propis del xip de RAM.

A nivell de seguretat i integritat, el TAD assegura que després de cada escriptura el bus de dades es torna a posar en mode lectura, per evitar col·lisions en cas que un altre mòdul intenti llegir de manera immediata. Aquesta precaució és essencial en un sistema cooperatiu on múltiples motors poden accedir a recursos compartits. Tot i això, cal remarcar que no es disposa d'un sistema per verificar la integritat de les dades (com ara checksums) ni cap mètode per identificar si una adreça concreta ha estat escrita prèviament, assumint un ús seqüencial i ordenat de les adreces per part del sistema.

Aquest TAD és principalment utilitzat pel **TAD_MENU** i el **TAD_CONTROLLER** per emmagatzemar els valors de temperatura i poder generar la gràfica dels valors acumulats Gràcies a la seva estructura modular i la seva integració acurada amb el maquinari, el **TAD_RAM** esdevé una peça fonamental per a l'emmagatzematge temporal de dades crítiques, mantenint alhora la simplicitat i compatibilitat amb el model cooperatiu que regeix tot el projecte.

TAD_RTC

El **TAD_RTC** és el responsable de la configuració, lectura i supervisió del rellotge de temps real del sistema, basat en el circuit DS3231. Aquest mòdul permet sincronitzar l'hora del sistema amb precisió i detectar quan transcorre un minut complet, fet essencial per a l'enviament regular d'actualitzacions horàries cap a la interfície Java. El TAD treballa íntegrament sobre el bus I2C, utilitzant funcions de lectura i escriptura no bloquejants a través del **TAD_I2C**, respectant així el model cooperatiu de la pràctica.

En primer lloc, la funció **RTC_Init()** s'encarrega de deixar el dispositiu en un estat inicial conegut. Per fer-ho, crida tres funcions internes: **configura_alarma1()**, **configura_INTCN_A1IE()** i **configura_limpiar_flags()**. Aquestes configuren l'alarma A1 perquè salti cada minut exactament quan el registre de segons val 00, activen les interrupcions internes del DS3231 i deixen el flag d'alarma net. Aquesta configuració garanteix que cada minut el RTC activarà un flag de manera automàtica que podrà ser detectat pel sistema.

Per configurar manualment l'hora i la data del RTC, es proporciona la funció **RTC_ConfigurarHora()**, que transforma els valors en format decimal a BCD (codificació Binari Codificat en Decimal) i escriu en cadena els valors als registres del DS3231 mitjançant una seqüència d'escriptura I2C. Aquesta funció també pot ser utilitzada durant el procés d'inicialització del sistema quan es rep una comanda **INITIALIZE** des de la interfície Java. Per recuperar l'hora i data actuals, s'utilitza la funció **RTC_readRTC()**, que llegeix seqüencialment els registres interns del dispositiu i torna els valors ja convertits de BCD a decimal per facilitar el seu ús dins del sistema.

El comportament dinàmic d'aquest TAD es basa en el motor **RTC_motor()**, una rutina cooperativa que s'ha de cridar periòdicament des del bucle principal. Aquesta funció accedeix al registre **0x0F** del RTC i comprova si el bit **A1F** (Alarm 1 Flag) ha estat activat. Si aquest bit està a 1 i no s'havia detectat prèviament, significa que ha transcorregut un minut complet, i per tant es crida **MENU_setAlarma(1)** per notificar-ho al **TAD_MENU**. Tot seguit, es neteja el flag del RTC mitjançant **configura_limpiar_flags()** i es guarda l'estat anterior del flag per evitar notificacions duplicades. Aquest mètode assegura una sincronització fiable i totalment cooperativa, ja que no depèn d'interrupcions del microcontrolador.

Aquest TAD manté una interacció directa amb diversos mòduls. Amb el **TAD_MENU** s'estableix la comunicació més estreta, ja que aquest rep la notificació de l'alarma de minut i reenvia la nova hora llegida a la interfície Java. També col·labora amb el **TAD_I2C**, del qual depenen totes les operacions de lectura i escriptura. Per garantir que les dades siguin interpretables pel sistema, es fan servir dues funcions auxiliars (**decimal_a_bcd()** i **bcd_a_decimal()**) que faciliten la conversió entre formats humans i la codificació requerida pel DS3231.

En conjunt, el **TAD_RTC** constitueix una peça fonamental per garantir el control temporal del sistema, proporcionant una base horària estable, precisa i accessible de manera no bloquejant. La seva integració amb el sistema garanteix una coordinació fluïda d'esdeveniments basats en el temps, com ara la generació de logs, actualització de l'hora o visualització cronològica de dades.

El format BCD (Binary Coded Decimal) utilitzat pel DS3231 codifica cada dígit decimal en grups de quatre bits, cosa que obliga a convertir els valors enviats o rebuts perquè puguin ser

manipulats de forma natural dins del sistema. Aquesta conversió es fa automàticament mitjançant les funcions auxiliars, garantint transparència i simplicitat en l'ús del RTC per part de la resta del codi.

Un altre aspecte rellevant és la gestió d'errors en les transmissions I2C. Cada funció que escriu al dispositiu retorna 0 si s'ha produït un error, com ara un mal reconeixement d'adreça o un dispositiu no present. En aquest cas, la comunicació s'atura immediatament amb una comanda **`I2C_Stop_()`** per evitar bloquejos o accés incorrecte al bus, afavorint una recuperació neta i controlada.

També cal destacar que la funció **`RTC_Init()`** només retorna èxit si les tres fases crítiques de configuració s'han completat correctament. Això proporciona una manera robusta i fiable de verificar si el dispositiu està connectat i operatiu en el moment d'arrencada del sistema.

Finalment, cal remarcar que tot el control de l'alarma es fa sense usar cap interrupció del microcontrolador. A diferència d'altres enfocaments basats en flags o vectors d'interrupció, aquest sistema revisa de manera periòdica l'estat del registre **`0x0F`**, el qual conté els bits d'estat del DS3231, entre ells l'**`A1F`**. Això garanteix que l'arquitectura es manté 100% cooperativa i compatible amb la resta de motors, mantenint una estructura clara, controlada i fàcilment depurable.

TAD_SIO

El **`TAD_SIO`** gestiona la comunicació UART entre el microcontrolador i la interfície externa (habitualment Java), utilitzant el mòdul EUSART del PIC18F4321 en mode asíncron. Aquest TAD proporciona una capa d'abstracció per facilitar l'enviament i la recepció de caràcters de manera fiable i compatible amb el model cooperatiu del sistema. És fonamental per garantir la interacció entre l'usuari i el sistema embegut, ja que la gran majoria de comandes i respostes passen per aquest canal.

La funció **`SIO_Init()`** configura els registres interns del mòdul EUSART per establir una velocitat de transmissió de 19200 bps amb un rellotge de sistema de 32 MHz. Per fer-ho, es desactiva la velocitat alta (**`BRGH = 0`**) i es configura el generador de baud rate en mode de 8 bits (**`BRG16 = 0`**), fixant **`SPBRG = 25`**. També s'activa la transmissió (**`TXEN = 1`**), la recepció contínua (**`CREN = 1`**) i l'habilitació general del port sèrie (**`SPEN = 1`**). Aquesta configuració és compatible amb la comunicació per bytes sense interrupcions, en mode totalment cooperatiu amb un error mínim.

Aquest TAD exposa quatre funcions bàsiques. La funció **`SIOTx_pucEnviar()`** consulta si el registre de transmissió està buit (**`TXIF = 1`**), indicant que es pot enviar un nou caràcter sense perdre dades. Si és així, la funció **`SIOTx_sendChar()`** escriu un nou byte al registre **`TXREG`**, que és transmès automàticament pel hardware del microcontrolador. Per a la recepció, **`SIORX_heRebut()`** comprova si hi ha un byte disponible al buffer de recepció (**`RCIF = 1`**), i en cas afirmatiu, **`SIORX_rebreChar()`** permet llegir el valor des de **`RCREG`**.

Aquestes funcions són utilitzades de manera extensiva des d'altres TADs com ara **`TAD_MENU`**, que estructura tota la seva comunicació amb Java basant-se en l'estat del registre UART. L'enviament de missatges es fa caràcter a caràcter, de forma controlada i sense bloquejos, evitant desbordaments de buffer i assegurant que cada byte és transmès correctament. El

mateix passa amb la recepció: es llegeix un caràcter cada vegada que el sistema detecta que el buffer està ple, integrant-se perfectament en el cicle de motors cooperatius.

La simplicitat del **TAD_SIO** és clau per la seva eficiència. No conté cap motor intern, ja que el seu funcionament es basa exclusivament en consultes puntuals a l'estat dels registres hardware. Aquest disseny el fa lleuger i altament reutilitzable, i facilita la seva integració en qualsevol punt del codi que necessiti enviar o rebre dades per UART. En sistemes embeguts amb recursos limitats, aquest enfocament garanteix una comunicació estable, previsible i fàcil de depurar.

En resum, el **TAD_SIO** actua com una interfície essencial entre el sistema i el món exterior, centralitzant tota la comunicació per sèrie amb una estructura mínima però funcional. La seva integració fluida amb el **TAD_MENU** i altres motors garanteix la coherència i robustesa de tot el sistema de missatgeria.

TAD_TEMP

El **TAD_TEMP** és el mòdul encarregat de gestionar la lectura periòdica del sensor analògic de temperatura del sistema. Està dissenyat per funcionar dins del model cooperatiu, adaptant la freqüència de mostreig a un valor que pot ser configurat dinàmicament per la interfície d'usuari mitjançant el **TAD_MENU**. Aquest mòdul permet obtenir temperatures amb una periodicitat ajustable i detectar quan hi ha una nova mostra disponible per a ser processada o enviada.

La funció **TEMP_initTemp()** inicialitza l'estat del TAD, col·locant-lo en **ESPERA_CONFIG**, estat que s'utilitza com a punt de bloqueig fins que es detecta que el sistema ha estat inicialitzat des del menú. També crea un temporitzador virtual (**tempTimer**) que s'utilitzarà per controlar el temps entre mostres. La freqüència de mostreig és gestionada mitjançant la variable **tempsMesura**, que es pot configurar a través de la funció **TEMP_setTempsMostreig()**, invocada habitualment per **TAD_MENU** quan es rep una comanda **INITIALIZE**.

La lògica principal es troba dins del motor cooperatiu **motorTemp()**, que implementa una màquina d'estats amb tres fases. En **ESPERA_CONFIG**, el sistema espera que el **TAD_MENU** indiqui que el sistema ha estat inicialitzat, mitjançant la funció **MENU_hiHaNewConfig()**. Quan això passa, s'entra a l'estat **LLEGINT_TEMP**, on es realitza una lectura ADC del sensor de temperatura i es converteix el valor digital a una temperatura expressada en graus Celsius mitjançant la funció **conversioADC_a_Temp()**. Aquesta funció utilitza una relació lineal per convertir el valor ADC (0-255) a una temperatura basada en un sensor TMP36, considerant una sortida de 500 mV a 0°C i un guany de 10 mV/°C.

Després de fer la conversió, el resultat es desa a **valor_temp**, es reinicia el temporitzador i s'activa el flag **hiHaNewTemp**, indicant que hi ha una nova mostra disponible. A continuació, es transita a **ESPERA_TEMPS**, on el sistema roman fins que el temporitzador indica que ha passat el temps de mostreig configurat. Llavors es torna a **LLEGINT_TEMP** i es repeteix el cicle. Durant aquest procés, **hiHaNewTemp** es posa a **TEMP_FALS** per defecte, i només es torna a activar després de cada lectura.

Per consultar si hi ha una nova mostra disponible, es pot invocar la funció **TEMP_hiHaNewTemp()**, la qual retorna **TEMP_CERT** si s'ha llegit una nova temperatura. D'altra banda, la funció **TEMP_getTemp()** retorna el valor de temperatura actual emmagatzemat a **valor_temp**, que pot ser utilitzat per altres mòduls com el de control o visualització.

Aquest TAD opera de manera completament no bloquejant, sincronitzat mitjançant el temporitzador virtual. No depèn d'interrupcions ni bloquejos actius, i permet que el sistema llegeixi de forma segura el sensor de temperatura amb una resolució i precisió suficient per a l'aplicació plantejada. És especialment útil en conjunció amb el **TAD_MENU** i el **TAD_LED**, que poden reaccionar als valors de temperatura actuals per mostrar informació o prendre accions correctives.

La funció de conversió utilitzada per obtenir la temperatura a partir del valor ADC es basa en una expressió aritmètica que té en compte que la lectura ADC oscil·la entre 0 i 255 amb una referència de 5V, cosa que dona una resolució aproximada de 19,6 mV per pas. Tenint en compte que el sensor TMP36 proporciona 500 mV a 0 °C i augmenta 10 mV per cada grau, això implica que cada pas representa prop de 2 °C. Aquesta resolució és limitada però suficient per a l'objectiu del sistema, i la fórmula aplicada permet una conversió directa, simple i computacionalment lleugera.

Cal tenir en compte que aquest TAD assumeix que el canal analògic corresponent al sensor està correctament configurat al sistema, especialment pel que fa als registres ADC del microcontrolador, i que el sensor està connectat de manera estable sense presència de soroll. A diferència de sistemes més avançats, aquesta lectura no es filtra ni es mitjana, fet que la pot fer sensible a fluctuacions puntuals. Tot i això, la simplicitat de l'esquema garanteix una implementació senzilla i efectiva dins d'un entorn cooperatiu.

Finalment, aquest TAD no només proporciona la temperatura al **TAD_LED** per canviar el color segons l'estat tèrmic, sinó que també actua com a font de dades per al **TAD_RAM** i **TAD_EEPROM**, els quals en registren els valors per a la seva posterior visualització en forma de gràfica o històric. Aquesta interconnexió amb altres TADs en fa una peça clau per a la monitorització contínua de l'estat del sistema.

TAD_VENTILADOR

El **TAD_VENTILADOR** és el mòdul encarregat de controlar la velocitat de dos ventiladors físics en funció del nivell tèrmic detectat pel sistema. Treballa de forma coordinada amb el **TAD_TEMP**, del qual rep la classificació tèrmica actual, i amb el **TAD_Config**, on es defineixen els valors simbòlics per a cada estat de temperatura i nivell de ventilació. Aquest mòdul gestiona els ventiladors mitjançant senyals digitals, generant una modulació PWM per controlar la velocitat en certs modes, i encenent o apagant els ventiladors directament en altres.

En la funció **VENT_InitVent()** s'inicialitzen els estats interns dels dos ventiladors a **VENT_CRITICAL**, es creen els temporitzadors virtuals corresponents i s'estableix la sortida inicial dels pins a **VENT_ESPERA**, deixant els dos ventiladors apagats. El sistema parteix del supòsit que una situació crítica implica la parada immediata dels ventiladors.

La funció **VENT_setVelocity()** rep un nivell tèrmic (per exemple, **TEMP_LOW**, **TEMP_MODERATE**, etc.) i actualitza l'estat intern dels dos ventiladors segons una matriu predefinida que relaciona directament el nivell de temperatura amb el comportament esperat de cada ventilador. Aquesta abstracció permet controlar la resposta del sistema amb una sola crida, mantenint encapsulada la lògica de traducció entre temperatura i velocitat dels ventiladors.

El motor cooperatiu **motorVentilador()** s'executa de manera independent per a cada ventilador (**v = VENT1** o **VENT2**) i actua en funció de l'estat actual assignat a cada un. Si l'estat és **VENT_LOW**, només el primer ventilador (**VENT1**) genera un senyal PWM, mentre que el segon es manté apagat. En **VENT_MODERATE**, ambdós ventiladors generen senyals PWM. En el mode **VENT_HIGH**, els dos ventiladors estan sempre activats (senyal constant a '1'), generant la màxima ventilació possible. Finalment, en **VENT_CRITICAL**, s'apaguen completament.

Els senyals PWM es generen de forma simple mitjançant un canvi d'estat (**toggle**) de la línia corresponent cada vegada que el temporitzador virtual indica que ha transcorregut el temps **VENT_PWM**. Això produeix un senyal quadrat amb cicle útil del 50%, que és suficient per generar una velocitat mitjana als ventiladors. Aquesta tècnica és compatible amb el model cooperatiu i no requereix cap perifèric de maquinari especialitzat, només temporitzadors virtuals i control directe de ports.

Gràcies a aquest disseny, el **TAD_VENTILADOR** permet una resposta tèrmica escalable i precisa amb una càrrega de càlcul molt baixa. La seva modularitat i separació clara d'estats el fan fàcilment ampliable i mantenible. A més, la seva coordinació amb el **TAD_TEMP** assegura que el comportament físic del sistema reflecteix fidelment l'estat tèrmic intern, garantint una ventilació eficient en tot moment.

Els valors associats a cada nivell de temperatura tenen un comportament pensat per optimitzar tant el consum energètic com la resposta davant situacions normals o crítiques. En el cas de **TEMP_LOW**, només un ventilador s'activa per reduir el soroll i el consum, mentre que **TEMP_CRITICAL** força l'aturada total dels ventiladors, entenent que en aquest escenari cal preservar el sistema o indicar un error greu, i no forçar components en un entorn potencialment insegur.

La funció **VENT_setVelocity()** és generalment cridada des del **TAD_CONTROLADOR** o des de l'estat adequat del **TAD_MENU**, segons el nivell tèrmic obtingut per **TAD_TEMP**. Això permet una resposta immediata i coordinada davant canvis de temperatura, contribuint a mantenir estable l'estat tèrmic general del sistema i protegir els components interns davant escalfaments.

Configuracions del microcontrolador

El sistema es basa en un microcontrolador PIC18F4321, la configuració del qual s'ha dut a terme tenint en compte la necessitat de simplicitat, robustesa i compatibilitat amb un entorn cooperatiu. Totes les configuracions bàsiques es defineixen mitjançant directives **`#pragma config`**, i la resta d'ajustos s'apliquen durant la fase d'inicialització a través de funcions específiques com **`InitOsc()`** i **`InitPorts()`**.

S'ha seleccionat l'oscil·lador intern com a font de rellotge (**`INTIO2`**) amb l'activació del PLL per obtenir una freqüència efectiva de funcionament de 32 MHz. Aquesta opció evita la necessitat de components externs com cristalls i ofereix una freqüència suficientment elevada per executar tots els motors del sistema de forma fluida. L'activació del PLL es realitza via programari amb **`OSCTUNEbits.PLLEN = 1`**, després d'establir el valor del registre **`OSCCON`**.

Pel que fa a la protecció i funcionalitat del microcontrolador, el Watchdog Timer (WDT) s'ha desactivat per evitar reinicis involuntaris. Així mateix, s'ha desactivat la programació en baixa tensió (**`LVP = OFF`**) i s'ha habilitat el pin de **`MCLR`** per disposar d'un mecanisme de reset extern. També es manté activada la protecció contra desbordaments de pila (**`STVREN = ON`**), per garantir un comportament segur en cas d'errors d'execució. Altres opcions com **`PBADEN = DIG`** s'han configurat per assegurar que els pins digitals B0-B4 funcionin com a tals des del primer moment, sense necessitat de reconfigurar-los a posteriori.

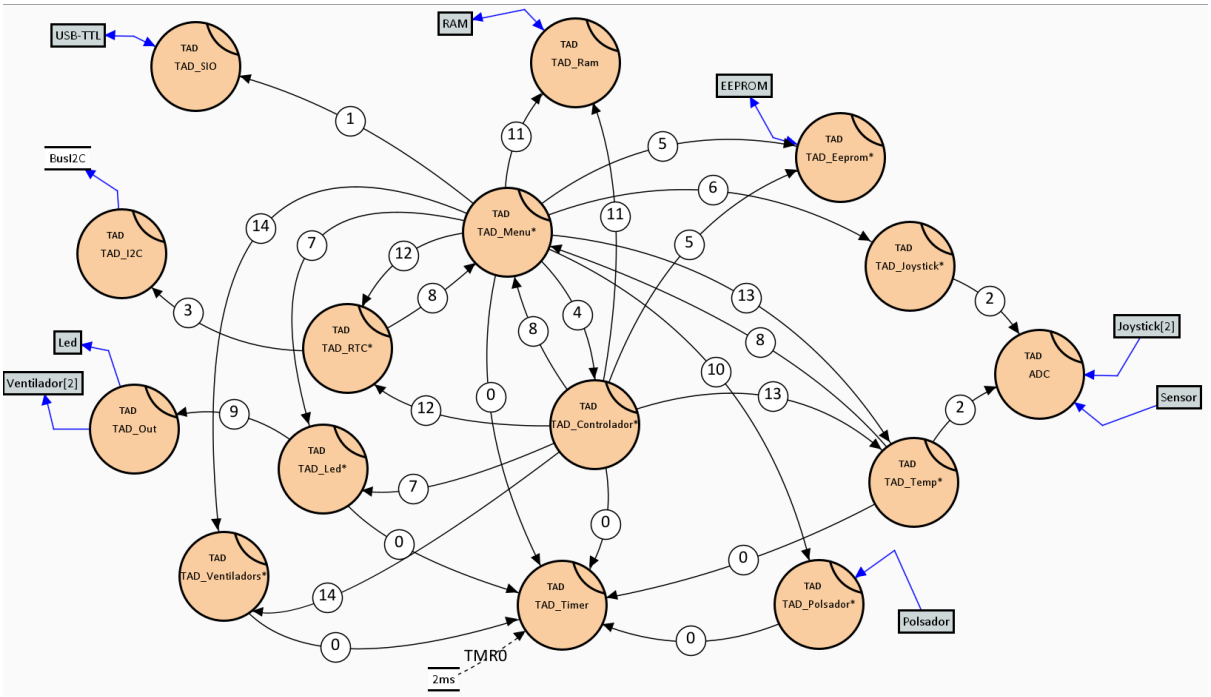
Tots els ports del sistema es configuren explícitament a la funció **`InitPorts()`**. Es defineixen com a entrades els pins analògics associats al sensor de temperatura (**`TRIS_TEMP`**), les dues components del joystick (**`TRIS_JOYX`**, **`TRIS_JOYY`**), el polsador (**`TRIS_POLS`**), els senyals de comunicació del RTC per I²C (**`TRIS_RTC_SCL`**, **`TRIS_RTC_SDA`**, **`TRIS_RTC_SQW`**), i la comunicació UART (**`TRIS_TX`**, **`TRIS_RX`**). S'estableixen com a sortides els pins dels tres canals del LED RGB (**`TRIS_LED_R`**, **`TRIS_LED_G`**, **`TRIS_LED_B`**), els dos ventiladors (**`TRIS_VENT1`**, **`TRIS_VENT2`**), i el pin **`RA6`** que serveix com a indicador de si el RTC s'ha inicialitzat correctament.

Un cop els ports estan preparats, s'inicialitzen els mòduls interns. El mòdul ADC permet obtenir lectures del sensor analògic TMP36, que són utilitzades pel **`TAD_TEMP`**. El mòdul EUSART és configurat per comunicació UART a 19200 bps, usat pel **`TAD_SIO`** per gestionar la comunicació amb la interfície gràfica. A més, s'activa el bus I²C per comunicar-se amb el RTC DS3231 mitjançant un conjunt de funcions pròpies no bloquejants definides al **`TAD_I2C`**. El sistema també fa ús del temporitzador TMR0, amb una rutina d'interrupció de prioritat alta que actualitza l'estat del temporitzador virtual, essencial per al funcionament cooperatiu.

Les interrupcions es configuren per funcionar amb el temporitzador TMR0, el qual ha estat inicialitzat amb el valor de recàrrega corresponent per generar una interrupció cada 2 mil·lisegons. Aquest període es calcula a partir de la configuració del registre **`T0CON`** i del valor **`RECARREGA_TMR0`**, que ha estat fixat a 49536 per obtenir l'interval desitjat amb una freqüència de rellotge de 32 MHz. L'ús d'aquesta interrupció periòdica permet incrementar un comptador global de tics, que és consultat pels diversos temporitzadors virtuals gestionats pel **`TAD_TIMER`**. Aquest mecanisme és la base del funcionament temporal de tots els motors cooperatius del sistema, assegurant una execució controlada, no bloquejant i precisa sense necessitat de múltiples fonts d'interrupció ni perifèrics addicionals.

Aquest conjunt de configuracions permet que el sistema operi de forma estable, previsible i sincronitzada. La configuració dels fuses assegura seguretat i control, mentre que la configuració dels ports i perifèrics garanteix que cada dispositiu associat (sensors, actuadors, busos de dades) pugui ser gestionat correctament pels TADs corresponents. Tot plegat constitueix una base sòlida per al desenvolupament d'un sistema modular, escalable i altament integrat.

Diagrama de TADs



Aquesta secció de la memòria té com a objectiu descriure el diagrama general de TADs (Tipus Abstractes de Dades) que estructuren el funcionament del sistema implementat. En aquest diagrama es representen les relacions de dependència entre mòduls, concretament quins TADs fan ús de les funcions d'altres TADs mitjançant crides directes. Cada fletxa del diagrama representa una dependència funcional, on l'origen de la fletxa és el TAD que realitza la crida, i el destí és el TAD cridat.

Per a facilitar la comprensió del diagrama, s'ha numerat cada fletxa amb el número d'interfície corresponent, seguint l'ordre de definició dels diccionaris d'interfície de cada TAD. També s'han identificat clarament les connexions amb perifèrics externs, com ara el joystick, el sensor de temperatura, la memòria RAM externa o el mòdul RTC, les quals es representen mitjançant fletxes blaves.

Tot seguit, es descriu detalladament la funcionalitat i les dependències de cada TAD, segons la informació extreta directament dels fitxers de codi font (.c).

TAD_ADC

Aquest TAD s'encarrega de la gestió del mòdul ADC (conversor analògic-digital) del microcontrolador. El seu motor intern (**ADC_motorADC**) fa servir una màquina d'estats que recorre cíclicament tres canals: temperatura (**CH_TEMP**), eix X del joystick (**CH_JOYX**) i eix Y del joystick (**CH_JOYY**). Per a cada canal, se selecciona el canal corresponent, s'inicia la conversió i, un cop completada, es guarda el valor corresponent en una variable interna (**valor_temp, valor_joyx, valor_joyy**).

Aquest TAD no rep dades d'altres TADs (cap fletxa entrant) i ofereix valors mitjançant tres funcions públiques (**ADC_getTemp**, **ADC_getJoyX** i **ADC_getJoyY**) que són utilitzades per TAD_Temp i TAD_Joystick, respectivament. Per tant, té dues fletxes sortints:

I. **TAD_Temp** (Interfície 13): fa servir **ADC_getTemp()** per obtenir la lectura del sensor de temperatura.

II. **TAD_Joystick** (Interfície 2): utilitza **ADC_getJoyX()** i **ADC_getJoyY()** per determinar la posició del joystick.

Aquest TAD no necessita cap inicialització externa un cop activat el sistema, ja que l'estat inicial queda establert a **ADC_InitADC()** i la seva funcionalitat principal recau en la crida periòdica al seu motor.

TAD_Controlador

Aquest TAD constitueix el nucli lògic del sistema, ja que coordina l'obtenció de dades ambientals, la seva classificació en funció dels llindars de temperatura, i la posterior acció sobre els dispositius de sortida. El seu motor **CONTROLLER_MotorControlador()** conté una màquina d'estats que comença en mode de configuració (**ESPERA_CONFIG**) i evoluciona cap a lectura de sensors (**TEMP_SENSOR**), gestió d'accions (**TEMP_RAM**, **TEMP_EEPROM**) i espera de nous mostreigs.

Aquest TAD presenta múltiples connexions sortints:

I. **TAD_Menu** (Interfície 8): crida funcions com **MENU_getTempBaixa**, **MENU_getTempAlta** o **MENU_hiHaNewConfig** per obtenir paràmetres inicials.

II. **TAD_Temp** (Interfície 13): crida **TEMP_getTemp()** per obtenir la lectura actual, i **TEMP_hiHaNewTemp()** per verificar si hi ha nova mostra.

III. **TAD_Led** (Interfície 7): utilitza **LED_setLedColor()** i **LED_setBlinkMode()** per indicar l'estat de temperatura.

IV. **TAD_Ventiladors** (Interfície 14): fa servir **VENT_setVelocity()** per controlar els ventiladors segons el llindar.

V. **TAD_Rtc** (Interfície 12): consulta l'hora actual amb **RTC_readRTC()** per registrar l'estat al moment de la mostra.

VI. **TAD_Eeprom** (Interfície 5): llegeix i escriu dades mitjançant funcions com **EEPROM_iniciaEscriptura**, **EEPROM_motorEscriptura**, **EEPROM_getContadorCadenes**, **EEPROM_getUltimaAddr**, etc.

VII. **TAD_Ram** (Interfície 11): emmagatzema les temperatures mitjançant **RAM_Write()** i reseteja l'adreça amb **Controller_clearRamAddr()**.

I una única entrada:

I. **TAD_Menu** (Interfície 8): quan el menú demana accions com la formateig o consulta de logs, utilitza **CONTROLLER_getLogFormatted()** o **Controller_getRamAddr()**.

Aquest TAD serveix com a pont central entre lectura, acció i registre, i per tant és un dels TADs amb més interaccions dins del sistema.

TAD_Eeprom

Aquest TAD encapsula tota la gestió de la memòria EEPROM del microcontrolador. Gestiona tant l'escriptura com la lectura mitjançant motors cooperatius: EEPROM_motorEscriptura i **EEPROM_motorLectura**. La memòria es tracta com un buffer cíclic de cadenes de 15 bytes, controlat per un comptador de cadenes i una adreça actual que es mantenen a les posicions 0 i 1 de la EEPROM.

Té sortides cap a:

I. **TAD_Menu** (Interfície 5): permet la lectura de logs des del menú mitjançant **EEPROM_resetEEPROM()** i altres funcions de lectura.

II. **TAD_Controlador** (Interfície 5): aquest TAD és qui principalment gestiona la memòria EEPROM per escriure logs en situació crítica, mitjançant **EEPROM_iniciaEscriptura()**, **EEPROM_motorEscriptura()**, **EEPROM_getUltimaAddr()**, etc.

No té entrades, ja que cap altre TAD depèn directament dels valors produïts pel TAD_Eeprom, sinó que només criden les seves funcions públiques.

Aquest mòdul incorpora funcions de control de flux (**EEPROM_esticEscriptura, EEPROM_lecturaCompleta**), funcions de gestió de cicles i reinicialització (**EEPROM_restaurarEstat, EEPROM_resetEEPROM**) i accés bàsic (**EEPROM_Llegeix, EEPROM_Escriu**).

TAD_JOYSTICK

Aquest TAD s'encarrega de determinar la posició actual del joystick a partir dels valors analògics llegits pels canals X i Y. El seu funcionament es basa en la lectura dels valors proporcionats pel **TAD_ADC** mitjançant les funcions **ADC_getJoyX()** i **ADC_getJoyY(),** que retornen valors entre 0 i 255. A partir d'aquests valors, el **TAD_Joystick** calcula si el joystick està en posició centrada o desplaçat cap a una de les direccions: amunt, avall, esquerra o dreta.

El motor **JOY_MotorJoystick()** és qui actualitza els valors interns de l'eix X i Y. Posteriorment, la funció **JOY_getPosicio()** retorna la direcció actual del joystick segons els llindars de tolerància definits.

Aquest TAD té una fletxa sortint:

I. **TAD_ADC** (Interfície 2): utilitza **ADC_getJoyX()** i **ADC_getJoyY()** per llegir els valors dels eixos del joystick.

Té també una fletxa entrant:

I. **TAD_Menu** (Interfície 8): és l'únic TAD que consulta la posició del joystick mitjançant JOY_getPosicio() per generar instruccions que es transmeten per UART.

Aquest TAD depèn per complet del **TAD_ADC** per obtenir les dades, i serveix com a mòdul intermedi per interpretar aquestes dades de forma simbòlica (direccions).

TAD_LED

Aquest TAD controla l'estat del LED RGB del sistema. Es permet encendre'l en un color determinat (vermell, verd o blau) o fer-lo parpellejar entre vermell i magenta si s'activa el mode de "blink". El comportament del LED s'organitza a través del motor cooperatiu **LED_motorLed()**, que implementa una màquina d'estats per gestionar tant la visualització fixa com el parpelleig.

El LED pot trobar-se en dos modes: en mode de color fix o en mode blink. En el primer cas, el LED mostra el color configurat mitjançant **LED_setLedColor()**, apagant prèviament la resta de components. En mode blink, el motor alterna automàticament entre encendre el LED en vermell i en magenta (vermell + blau) cada 500 tics de temporitzador (1 segon total per cicle complet), usant el **TAD_Timer**.

Aquest TAD té dues fletxes sortints:

I. **TAD_Timer** (Interfície 0): fa servir **TI_NewTimer()**, **TI_GetTics()** i **TI_ResetTics()** per controlar el temps de parpelleig.

II. **TAD_Out** (Interfície 9): utilitza **OUT_EncenLed()** i **OUT_ApagaLed()** per activar els colors corresponents.

I rep dues fletxes entrants:

I. **TAD_Controlador** (Interfície 7): li indica el color a encendre o si ha d'entrar en mode blink.

II. **TAD_Menu** (Interfície 8): el reinicialitza mitjançant **LED_InitLed()** en les operacions de reset.

Aquest TAD no interactua amb perifèrics externs de manera directa, sinó que actua com a capa d'abstracció sobre el **TAD_Out**, responsable final de la manipulació dels pins.

TAD_MENU

Aquest TAD és el responsable de la interacció entre el sistema embegut i la interfície gràfica JAVA. Opera com a gestor principal de comandes de l'usuari i transmet l'estat del sistema cap a l'exterior mitjançant UART. El seu motor, **MENU_MotorMenu()**, és una màquina d'estats que gestiona l'entrada del joystick i del polsador, la recepció de cadenes per UART, i la resposta corresponent segons el contingut de la comanda rebuda.

Aquest TAD pot enviar instruccions per UART com moviments del joystick, resposta a instruccions com **GET_LOGS**, **GET_GRAPH** o **RESET**, i enviar l'hora actualitzada a petició. També permet actualitzar paràmetres interns com els llindars de temperatura o el temps de mostreig mitjançant comandes com **INITIALIZE** o **SET_TIME**.

Té les següents fletxes sortints:

I. **TAD_Sio** (Interfície 1): mitjançant **SIOTx_sendChar()** i **SIORX_rebreChar()** per la comunicació UART.

II. **TAD_Joystick** (Interfície 2): consulta **JOY_getPosicio()** per saber el moviment del joystick.

III. **TAD_Polsador** (Interfície 10): consulta si s'ha premut el polsador amb **POLSADOR_HiHaPolsacio()**.

IV. **TAD_Timer** (Interfície 0): fa servir **TI_NewTimer()**, **TI_GetTics()** i **TI_ResetTics()** per temporitzacions internes.

V. **TAD_Rtc** (Interfície 12): consulta l'hora actual amb **RTC_readRTC()** i actualitza l'hora amb **RTC_ConfigurarHora()**.

VI. **TAD_Eeprom** (Interfície 5): reinicia la EEPROM amb **EEPROM_resetEEPROM()**.

VII. **TAD_Ram** (Interfície 11): consulta valors mitjançant **RAM_Read()** i reinicia la memòria amb **RAM_resetRAM()**.

VIII. **TAD_Controlador** (Interfície 4): consulta logs amb **CONTROLLER_getLogFormatted()** i adreces amb **Controller_getRamAddr().**

IX. **TAD_Led** (Interfície 7): reinicialitza el LED amb **LED_InitLed()**.

X. **TAD_Temp** (Interfície 13): reinicialitza el sensor amb **TEMP_initTemp()**.

XI. **TAD_Ventilador** (Interfície 14): reinicialitza els ventiladors amb **VENT_InitVent()**.

També rep entrades de:

I. **TAD_Rtc**: que li comunica l'activació de l'alarma mitjançant **MENU_setAlarma()**.

Aquest TAD és central en el sistema, ja que coordina les accions segons les ordres externes, i és l'únic TAD que interactua de forma dinàmica amb l'usuari mitjançant UART.

TAD_OUT

Aquest TAD s'encarrega de controlar directament els pins de sortida digitals connectats als LEDs RGB. A diferència del **TAD_Led**, que gestiona el comportament lògic i els estats com el parpelleig o els canvis de color, el **TAD_Out** és responsable d'encendre i apagar físicament els LEDs corresponents escrivint sobre els registres LAT del microcontrolador.

Aquest mòdul implementa tres funcions públiques:

I. **OUT_Init()**: que posa a 0 els pins dels LEDs a l'inici del sistema.
II. **OUT_EncenLed(unsigned char led)**: que activa el pin corresponent al color especificat.
III. **OUT_ApagaLed(unsigned char led)**: que apaga el pin corresponent.

Té una única fletxa entrant:

I. **TAD_Led** (Interfície 9): és l'únic TAD que utilitza les funcions d'aquest mòdul per encendre i apagar els LEDs segons la lògica que determina el motor de **LED_motorLed()**.

Aquest TAD no depèn de cap altre i no inclou cap lògica ni motor propi, ja que actua exclusivament com a capa d'abstracció hardware.

TAD_POLSADOR

Aquest TAD gestiona la lectura d'un polsador connectat a una entrada digital del microcontrolador, incloent-hi la detecció fiable de pulsacions mitjançant una FSM amb control de rebots (debouncing). Utilitza un temporitzador per assegurar que els canvis en el senyal són estables abans de considerar-los com a polsacions vàlides.

El motor cooperatiu **POLSADOR_MotorPolsador()** és qui executa la màquina d'estats. Aquesta passa pels estats: **POLS_NO_PREMUT**, **POLS_REBOTS_IN**, **POLS_PREMUT** i **POLS_REBOTS_OUT**, garantint que només es detectin pulsacions reals. Quan una pulsació és detectada, s'activa un flag intern que pot ser consultat amb la funció **POLSADOR_HiHaPolsacio()**.

Té una fletxa sortint:

I. **TAD_Timer** (Interfície 0): fa servir **TI_NewTimer()**, **TI_GetTics()** i **TI_ResetTics**() per controlar el temps de rebot.

Té una fletxa entrant:

I. **TAD_Menu** (Interfície 8): consulta l'estat del polsador amb **POLSADOR_HiHaPolsacio()** per canviar l'estat del sistema o activar l'enviament d'una comanda.

Aquest TAD opera de manera independent un cop inicialitzat, i la seva funció principal és informar de manera robusta sobre la interacció física de l'usuari amb el sistema.

TAD_RAM

Aquest TAD gestiona l'accés a la memòria RAM externa (tipus 62256) connectada al microcontrolador. Inclou funcions per escriure (**RAM_Write**), llegir (**RAM_Read**) i reinicialitzar (**RAM_resetRAM**) dades a la memòria. Aquesta RAM s'utilitza per emmagatzemar temperatures de forma contínua durant el funcionament del sistema.

L'accés a la RAM es fa mitjançant un sistema de multiplexació entre les adreces baixes (port B) i les adreces altes (enviades a través de latches), i un bus de dades bidireccional per port D. La sincronització amb els latches d'adreces altes es fa mitjançant un pols al senyal de clock (**RA7**).

Té una fletxa sortint:

I. **TAD_Menu** (Interfície 11): llegeix dades amb **RAM_Read()** per generar gràfiques i reinicia la memòria amb **RAM_resetRAM()**.

Té una fletxa entrant:

I. **TAD_Controlador** (Interfície 11): escriu temperatures periòdicament amb **RAM_Write()**, i consulta o reinicia l'adreça actual.

Aquest TAD no té motor propi, ja que totes les operacions es fan sota demanda a través de les seves funcions, sovint de forma cooperativa des d'altres TADs. Proporciona una memòria de registre de dades d'accés ràpid per al sistema.

TAD_RTC

Aquest TAD s'encarrega de la comunicació amb el rellotge en temps real (RTC) DS3231 a través del protocol I2C. Permet configurar l'hora (**RTC_ConfigurarHora**), llegir-la (**RTC_readRTC**) i controlar l'alarma per minut a través del motor cooperatiu **RTC_motor**. Aquest motor consulta de manera cíclica si ha saltat l'alarma del RTC llegint el registre 0x0F, i en cas afirmatiu, esborra el flag i avisa el **TAD_Menu** per mitjà de la funció **MENU_setAlarma().**

Aquest TAD presenta dues fletxes sortints:

I. **TAD_I2C** (Interfície 6): fa servir funcions com **StartI2C()**, **I2C_Write()**, **I2C_Read()**, **ReStartI2C_()** i **I2C_Stop_()** per accedir als registres del DS3231.

II. **TAD_Menu** (Interfície 12): el motor crida **MENU_setAlarma(1)** quan detecta que ha saltat l'alarma per minut.

També té dues fletxes entrants:

I. **TAD_Controlador** (Interfície 12): consulta l'hora actual mitjançant **RTC_readRTC()** per poder incloure-la als logs.

II. **TAD_Menu** (Interfície 12): crida **RTC_ConfigurarHora()** quan l'usuari fa un **SET_TIME** a través de la interfície JAVA.

Aquest TAD és fonamental per mantenir la traçabilitat temporal de les dades guardades i per sincronitzar accions com la transmissió de l'hora o la comprovació d'alarmes cícliques.

TAD_SIO

Aquest TAD gestiona la comunicació sèrie UART (EUSART) per al microcontrolador, permetent la transmissió i recepció de caràcters amb la interfície externa, que en aquest cas és una aplicació JAVA. Implementa funcions bàsiques com **SIOTx_sendChar()** per transmetre un caràcter, **SIORX_rebreChar()** per llegir-ne un rebut, i les funcions **SIOTx_pucEnviar()** i **SIORX_heRebut()** per comprovar l'estat dels buffers.

Aquest TAD no disposa de motor intern ni màquina d'estats. El seu funcionament es basa en l'accés immediat i no bloquejant al hardware UART. Té una funció d'inicialització, **SIO_Init()**, que configura els registres EUSART a 9600 bps, asíncron i amb **BRG** de 8 bits.

Té una fletxa entrant:

I. **TAD_Menu** (Interfície 1): utilitza la UART com a canal principal de comunicació amb la interfície gràfica externa. Fa servir **SIOTx_sendChar()**, **SIOTx_pucEnviar()**, **SIORX_heRebut()** i **SIORX_rebreChar()** per enviar ordres i rebre respostes o instruccions.

Aquest TAD no depèn de cap altre i té una funcionalitat essencial per al sistema, ja que constitueix el pont entre l'usuari i el dispositiu embegut.

TAD_TEMP

Aquest TAD s'encarrega de gestionar la lectura de la temperatura ambiental mitjançant el sensor TMP36 connectat al canal analògic del microcontrolador. El seu motor cooperatiu, **motorTemp()**, és una màquina d'estats que espera una nova configuració (via **TAD_Menu**), llegeix periòdicament una mostra mitjançant l'ADC i la converteix a graus Celsius. La lectura es fa amb la funció **ADC_getTemp()**, i la conversió es basa en una transformació lineal.

També gestiona el temporitzador associat a la freqüència de mostreig definida per l'usuari, que s'ajusta a través de **TEMP_setTempsMostreig()**. Aquest sistema garanteix que les lectures es facin a intervals regulars, controlats pel **TAD_Timer**. Quan una nova lectura és disponible, s'activa un flag consultable amb **TEMP_hiHaNewTemp()**.

Té les següents fletxes sortints:

I. **TAD_Timer** (Interfície 0): fa servir **TI_NewTimer()**, **TI_GetTics()** i **TI_ResetTics()** per temporitzar el mostreig.

II. **TAD_ADC** (Interfície 3): utilitza **ADC_getTemp()** per obtenir la lectura analògica del sensor.

També té fletxes entrants de:

I. **TAD_Co**ntrolador (Interfície 13): consulta **TEMP_getTemp()** i **TEMP_hiHaNewTemp()** per saber el valor actual i si s'ha fet una nova lectura.

II. **TAD_Menu** (Interfície 13): reinicialitza el sistema de temperatura amb **TEMP_initTemp()**.

Aquest TAD actua com a pont entre el món físic i la lògica del sistema, proporcionant lectures fiables i sincronitzades del sensor ambiental.

TAD_VENTILADOR

Aquest TAD controla el funcionament de dos ventiladors que regulen la temperatura del sistema. Cada ventilador pot estar en un dels següents estats: **VENT_LOW**, **VENT_MODERATE**, **VENT_HIGH** o **VENT_CRITICAL**. Segons l'estat assignat, el motor **motorVentilador()** activa o desactiva els ventiladors amb senyals digitals o PWM simulat, utilitzant un temporitzador propi per a cada un.

El **TAD_Ventilador** no implementa una lògica de decisió pròpia, sinó que actua com a executor de les ordres de temperatura provinents d'altres TADs. Cada ventilador es controla de manera independent mitjançant una crida periòdica al motor amb l'índex corresponent.

Té una fletxa sortint:

I. **TAD_Timer** (Interfície 0): utilitza **TI_NewTimer()**, **TI_GetTics()** i **TI_ResetTics**() per generar els senyals PWM.

Té una fletxa entrant:

I. **TAD_Controlador** (Interfície 14): li assigna el nivell de funcionament mitjançant **VENT_setVelocity()**, i invoca el motor cooperatiu **motorVentilador()**.

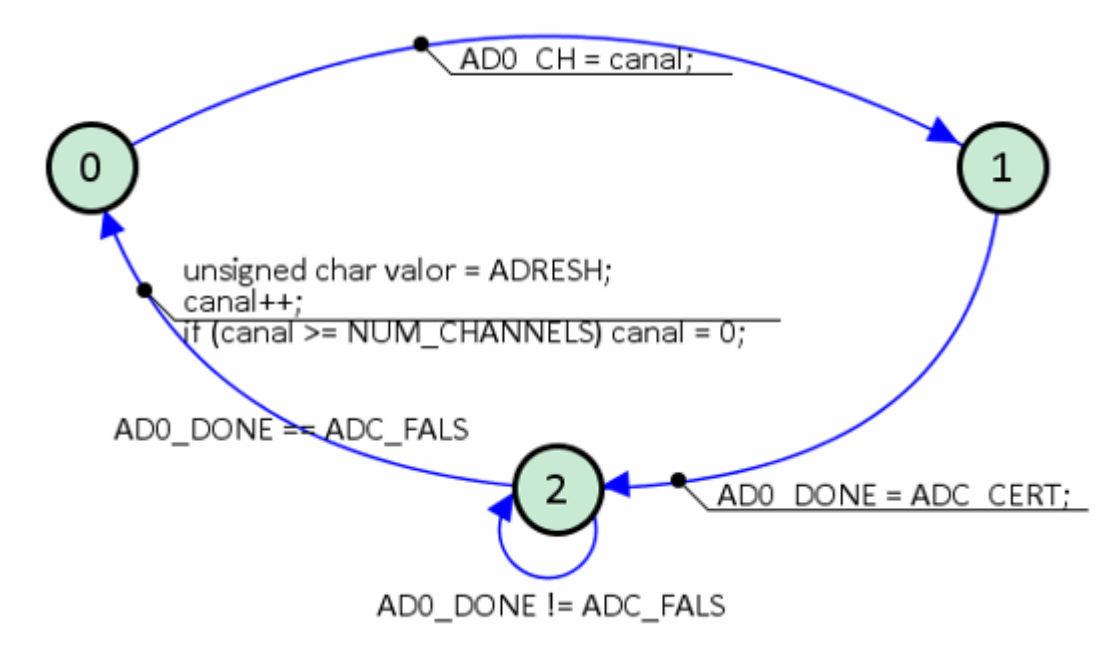
També és reinicialitzat per:

I. **TAD_Menu** (Interfície 14): amb la crida a **VENT_InitVent()** durant les operacions de RESET.

Aquest TAD actua com a mòdul de sortida directa sobre els ventiladors i garanteix un comportament dinàmic i controlat segons la temperatura del sistema.

Motors dels TADs

MOTOR ADC

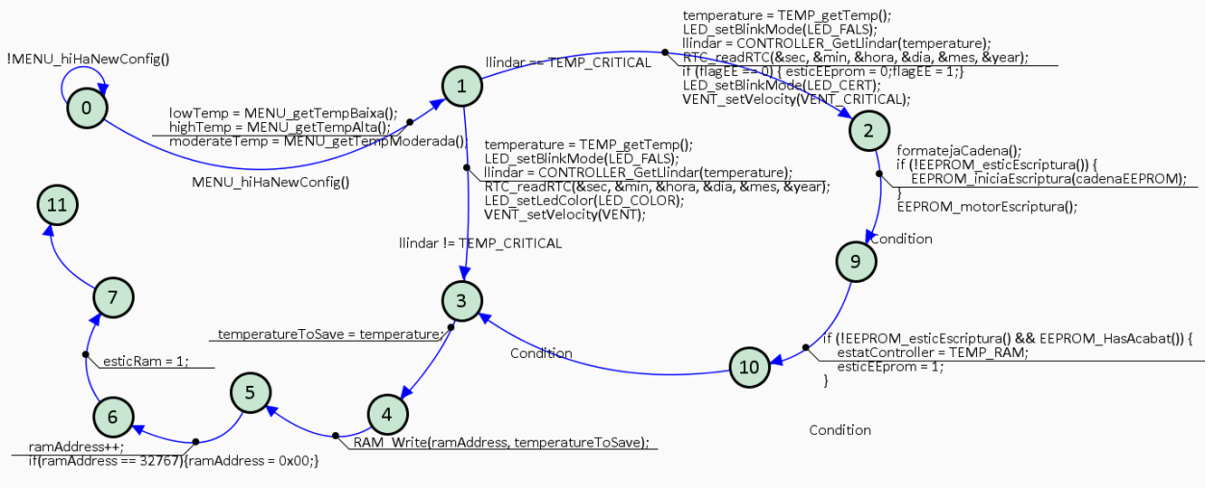


El motor cooperatiu encarregat de gestionar el mòdul ADC s'encarrega de llegir cíclicament els valors de tres canals analògics: els dos corresponents a les coordenades del joystick i un tercer assignat al sensor de temperatura. Aquest motor, implementat com una màquina d'estats finits, segueix una estructura composta per tres estats principals que permeten controlar el procés de conversió sense bloquejar l'execució global del sistema.

En l'estat inicial, el motor selecciona el canal que toca llegir mitjançant l'assignació al registre **AD0_CH**, on canal és una variable que es va incrementant a cada iteració. A continuació, el motor passa a un segon estat on s'inicia la conversió analògic-digital, i entra en un tercer estat d'espera. En aquest darrer estat, es comprova de manera regular si la conversió ha finalitzat observant el flag **AD0_DONE**. Mentre aquest flag no és actiu, el motor es manté a l'espera de manera no bloquejant, permetent que la resta del sistema continuï executant-se. Quan **AD0_DONE** indica que la conversió ha finalitzat, es llegeix el valor de **ADRESH**, que conté la dada digitalitzada del canal actiu, i es torna a l'estat inicial. Abans de tancar el cicle, s'incrementa el comptador de canal i, si cal, es torna a zero per mantenir el cicle circular de lectures.

Aquesta estratègia assegura una distribució equitativa de les conversions entre canals i evita colls d'ampolla que podrien aparèixer si es prioritzés un canal sobre els altres. L'ús del registre de resultat **ADRESH** permet obtenir la part significativa de la conversió amb rapidesa, mentre que la verificació de **AD0_DONE** evita l'ús d'espera activa, característica que fa compatible aquest motor amb l'arquitectura cooperativa del sistema. Així, es garanteix que el **TAD_JOYSTICK** i el **TAD_TEMP** disposin sempre d'un valor de lectura actualitzat dins d'un marge temporal previsible, sense comprometre la reactivitat del conjunt del sistema.

MOTOR CONTROLLER



El motor cooperatiu del **TAD_CONTROLLER** és una de les peces clau del sistema, ja que és el responsable de supervisar la temperatura ambiental i prendre accions tant de control com de registre. El seu disseny segueix una màquina d'estats finits força extensa i ben estructurada, formada per 12 estats que permeten tractar de manera diferenciada cada etapa del procés de control i registre, evitant bloquejos i garantint una execució fluïda en entorns cooperatius.

En l'estat 0, el motor comprova si el sistema ha estat inicialitzat mitjançant **MENU_hiHaNewConfig()**. Si encara no ho ha estat, es manté en aquest estat. Un cop detectada la inicialització, es llegeixen els llindars de temperatura configurats des del menú i s'avança cap a l'estat 1. Aquí, es fa una primera lectura de temperatura, es comprova si aquesta supera el llindar crític, i si és així, es configura el LED en mode de parpelleig i s'activa el mode crític de ventilació. Aquesta decisió també activa la variable **flagEE** i bloqueja l'escriptura de dades a EEPROM i RAM per evitar registrar valors en situacions extremes.

Si la temperatura no és crítica, es passa a l'estat 3, on es torna a llegir la temperatura i es calcula el nivell tèrmic corresponent mitjançant **CONTROLLER_GetLlindar()**. També s'obté l'hora del RTC i es configuren tant el color del LED com la velocitat dels ventiladors en funció del nivell detectat. Aquesta divisió entre estat crític i no crític permet al sistema actuar de manera segura i eficient segons la situació tèrmica del moment.

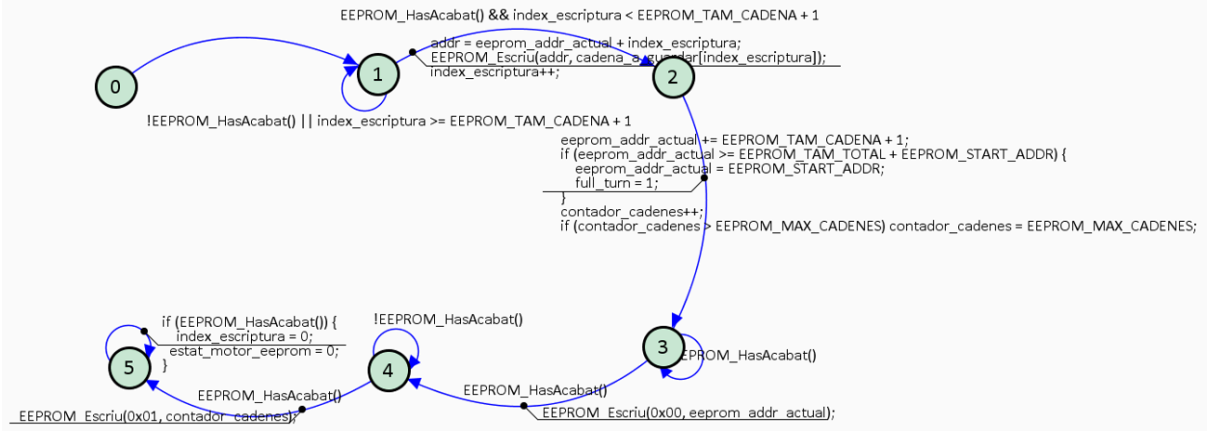
L'estat 2 és utilitzat exclusivament per a la gestió del mode crític. Si s'ha activat aquest mode, el motor activa el flag **LED_CERT**, llança **VENT_setVelocity()** amb l'estat **CRITICAL**, i després salta directament a l'estat 11 per tancar el cicle sense fer escriptures.

En condicions normals, es continua per l'estat 4, on es guarda la temperatura actual en una variable temporal. En l'estat 5 es prepara l'adreça de la RAM on s'haurà d'escriure, i en l'estat 6 s'incrementa aquesta adreça, comprovant que no superi el límit de capacitat (en aquest cas, 32767). Si ho fa, es reinicia a zero per mantenir el funcionament cíclic.

Els estats 7 a 10 gestionen l'escriptura a l'EEPROM. Primer s'activa el flag **esticRam = 1**, indicant que caldrà guardar la dada també a RAM. En l'estat 9 es comprova si l'EEPROM està lliure (**!EEPROM_esticEscriptura()**) i si cal, s'inicia una nova escriptura mitjançant **EEPROM_iniciaEscriptura()**. L'estat 10 espera que aquesta finalitzi. Un cop ha acabat, es permet passar a RAM, amb l'estat de control **TEMP_RAM.**

Amb tot el procés completat, el motor passa a l'estat 11, des d'on retorna a l'estat inicial i espera una nova iteració. Aquest funcionament segmentat, on cada acció ocupa un cicle d'execució diferent, assegura que les operacions com la lectura de sensors, el control dels actuadors i l'escriptura de dades persistents no interfereixin entre si i es puguin repartir de manera equitativa dins el temps disponible del sistema.

MOTOR EEPROM



El **TAD_EEPROM** consta de dos motors, un de lectura, i un d'escriptura.

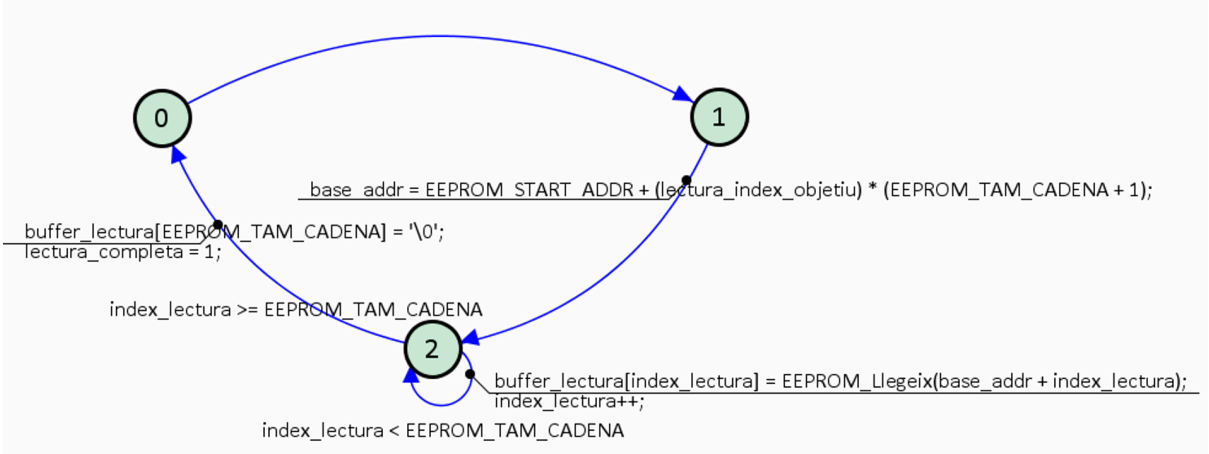
El **motorEscriptura()** és l'encarregat de gestionar l'escriptura persistent de cadenes de text a la memòria EEPROM del sistema, mantenint una estructura circular que permet conservar un nombre limitat de registres i evitar el desbordament de memòria. Està organitzat com una màquina d'estats de sis fases que reparteixen de manera cooperativa tota l'operació d'escriptura, evitant qualsevol tipus de bloqueig o espera activa.

En l'estat inicial, el motor comprova si encara queden caràcters per escriure a la cadena i si la memòria està disponible. Aquesta verificació es fa amb el flag **EEPROM_HasAcabat()** i el comptador **index_escriptura**, que marca la posició actual dins de la cadena. Si es compleixen les condicions, es passa a l'estat 1, on es calcula l'adreça física a la qual s'ha d'escriure i es realitza l'escriptura d'un sol caràcter mitjançant **EEPROM_Escriu()**. Tot seguit, es torna al pas inicial incrementant l'índex per continuar amb el caràcter següent.

Quan s'ha completat la cadena, s'entra a l'estat 2, on s'actualitza el punter global **eeprom_addr_actual** per avançar fins a la següent posició d'inici de cadena. Si aquest punter ha arribat al final de l'espai reservat a EEPROM, es reinicia a **EEPROM_START_ADDR** i es marca amb **full_turn = 1** que s'ha completat una volta completa. També es manté i incrementa un comptador de cadenes (**contador_cadenes**) fins a arribar al màxim definit. A partir d'aquest punt, qualsevol nova cadena sobreescriurà la més antiga, implementant un comportament de buffer circular.

L'estat 3 comprova si la memòria està lliure per poder escriure el valor 0x00 a la posició inicial de la nova cadena, una acció que serveix com a marcatge de separació entre cadenes. Aquesta escriptura es fa també mitjançant **EEPROM_Escriu()**, i el sistema espera a l'estat 4 fins que el dispositiu confirma que ha finalitzat la seva operació interna.

Finalment, en l'estat 5, un cop tot el procés ha estat completat, es torna a posar a zero l'índex d'escriptura i es reinicia l'estat del motor a 0, deixant-lo preparat per a la propera cadena. Aquesta segmentació en múltiples estats assegura que l'escriptura es reparteixi en diversos cicles de processament, evitant saturar el sistema i garantint la compatibilitat amb altres motors que comparteixen l'ús de la UART o altres perifèrics interns.



El **motorLectura()** és l'encarregat de gestionar la lectura no bloquejant de cadenes de text emmagatzemades a la memòria EEPROM, permetent accedir de manera controlada a les dades registrades prèviament per l'usuari. Està dissenyat com una màquina d'estats simple però eficaç, que divideix el procés de lectura en fases cooperatives per garantir la compatibilitat amb altres motors concurrents dins del sistema.

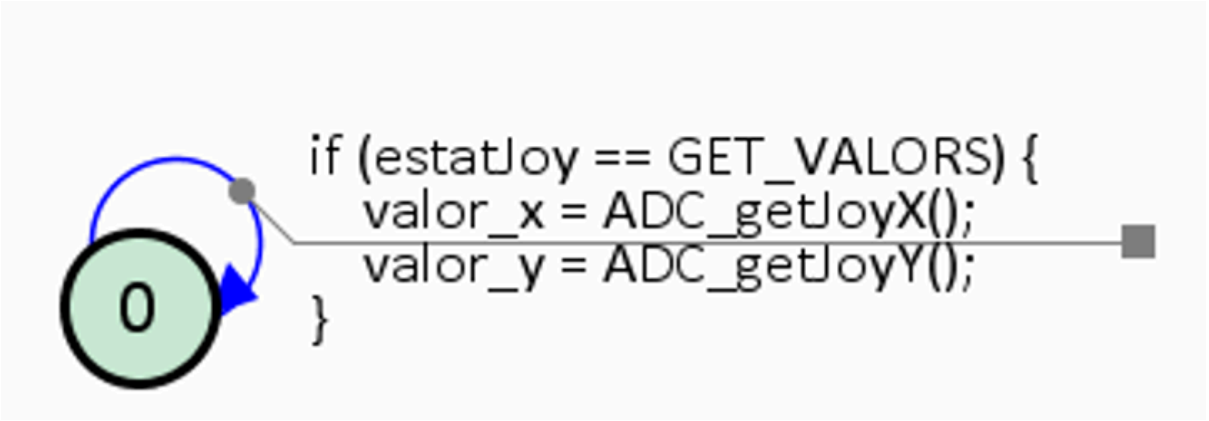
La lectura s'inicia quan el sistema estableix un valor objectiu a la variable **lectura_index_objetiu**, que representa l'índex ordinal de la cadena que es vol recuperar. El motor comença en l'estat 1, on calcula l'adreça base de la cadena a partir de la posició d'inici reservada a EEPROM (**EEPROM_START_ADDR**), sumant-hi el desplaçament corresponent al nombre de caràcters de cada cadena (definit per **EEPROM_TAM_CADENA + 1**, incloent el caràcter de separació o finalització). Aquesta adreça es desa a la variable interna **base_addr**.

Un cop obtinguda l'adreça inicial, el motor passa a l'estat 2, que és el nucli del procés de lectura. En aquest estat, i mentre no s'hagin llegit tots els caràcters previstos, es fa una lectura seqüencial de la memòria EEPROM posició a posició, copiant el contingut directament al buffer de lectura **buffer_lectura**. Aquest buffer és l'encarregat de contenir la cadena llegida per poder-la utilitzar posteriorment dins del sistema.

El progrés es controla mitjançant el comptador **index_lectura**, que incrementa en cada iteració i indica la posició actual dins la cadena. Quan aquest comptador arriba al límit definit per **EEPROM_TAM_CADENA**, el motor escriu manualment el caràcter de terminació **'\0'** per assegurar que la cadena pugui ser tractada correctament com una cadena de text en C. A continuació, es marca la lectura com a completada mitjançant el flag **lectura_completa = 1**, i es reinicia l'estat del motor a 0, deixant-lo preparat per a una nova operació de lectura.

El motor del TAD_EEPROM ofereix així una solució elegant i eficient per enregistrar dades textuals de manera persistent, mantenint l'ordre d'arribada i assegurant una rotació controlada de l'espai de memòria disponible.

MOTOR JOYSTICK



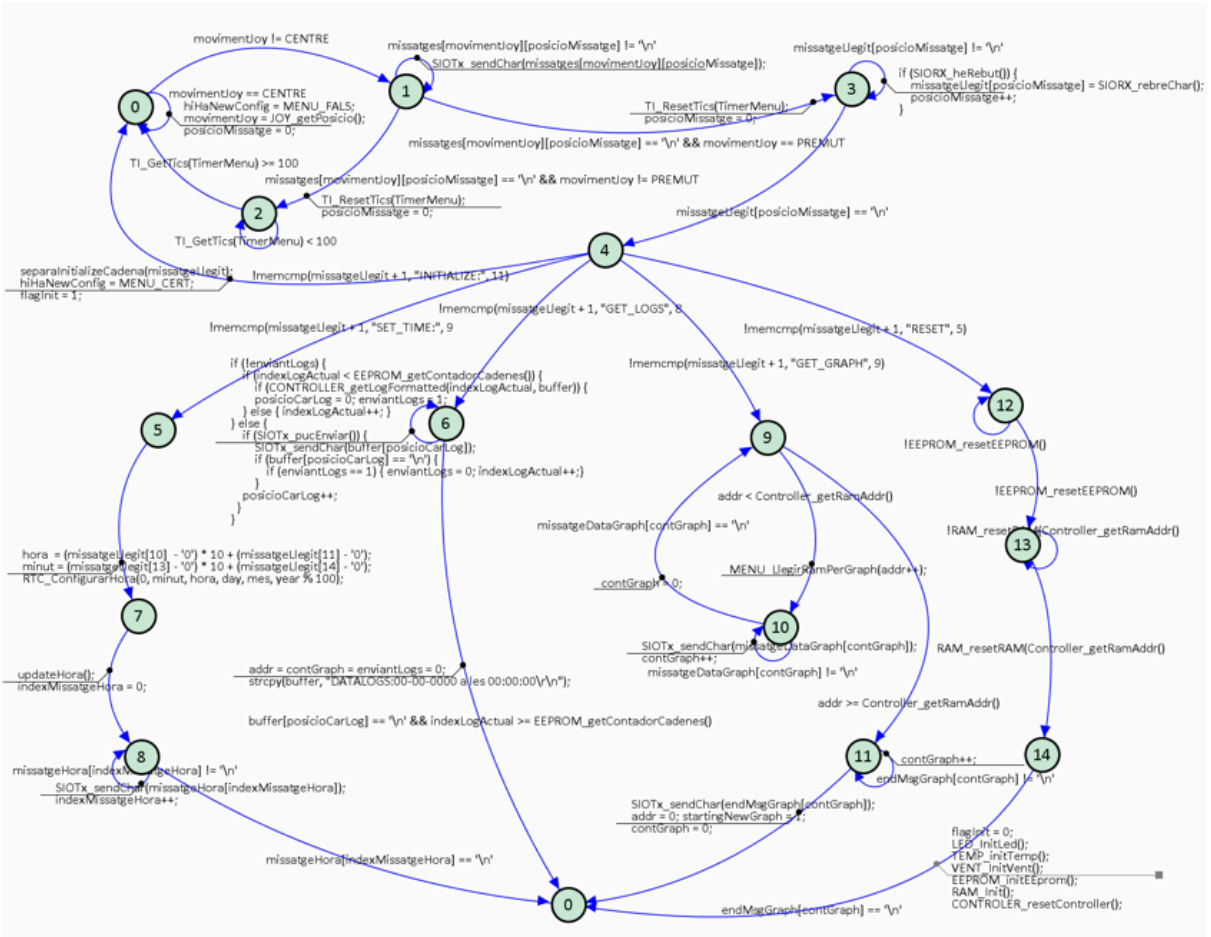
El motor associat al **TAD_JOYSTICK** és un dels més simples del sistema, però compleix una funció essencial per a la navegació dins del menú. Està implementat com una màquina d'un sol estat que es limita a actualitzar de manera periòdica les dues coordenades del joystick, utilitzant valors obtinguts directament del **TAD_ADC**. Tot i la seva simplicitat estructural, aquest motor garanteix la reactivitat immediata del sistema a l'entrada de l'usuari.

En cada iteració del bucle principal, el motor entra en execució i comprova si es troba en l'estat **GET_VALORS**, que és l'únic que té implementat. En cas afirmatiu, es criden les funcions **ADC_getJoyX()** i **ADC_getJoyY()**, les quals retornen els últims valors llegits pel canal d'ADC corresponent als eixos horitzontal i vertical del joystick. Aquests valors es desen a les variables **valor_x** i **valor_y**, que es troben declarades dins del **TAD_JOYSTICK** i que posteriorment són consultades per la funció **JOY_getPosicio()** per determinar la direcció actual del moviment.

La simplicitat d'aquest motor fa que sigui altament eficient i que s'executi sense retard dins del model cooperatiu. No utilitza temporitzadors ni flags intermedis, i depèn exclusivament de les lectures actualitzades que proporciona el **TAD_ADC**. Gràcies a aquest funcionament directe i immediat, el sistema és capaç de detectar els canvis en la posició del joystick amb una latència mínima, la qual cosa permet una navegació suau i precisa pel menú.

Tot i estar format per un únic estat i no tenir transicions, aquest motor compleix perfectament amb el principi de no bloqueig i s'integra a la perfecció amb la resta de motors del sistema. La seva presència constant al bucle principal assegura que qualsevol moviment fet per l'usuari serà processat gairebé de manera instantània.

MOTOR MENU



El motor del **TAD_MENU** és un dels més extensos i centrals del sistema, ja que actua com a nexe entre la interfície d'usuari (Java) i la lògica interna del microcontrolador. Controla tant l'enviament de moviments del joystick com la recepció i interpretació de comandes, i a més és responsable de gestionar els processos de configuració inicial, actualització de l'hora, peticions de gràfiques, logs i reset del sistema. La seva implementació es basa en una màquina d'estats que gestiona amb precisió el diàleg bidireccional amb la interfície, mantenint sempre la cooperativitat.

En l'estat **ESTAT_JOY**, el sistema detecta el moviment actual del joystick mitjançant **JOY_getPosicio()** i prepara l'enviament del missatge associat si el moviment no és el de repòs (**CENTRE**). Si es detecta moviment, es passa a l'estat **ENVIANT_JOY**, on es transmet caràcter a caràcter la cadena corresponent (**UP, DOWN,** etc.) per UART. Quan s'arriba al final de la cadena (**\n**), es reinicia el temporitzador i, en funció de si el moviment ha estat una pulsació (**PREMUT**), es transita a **ESTAT_LLEGEIX** o a **ESTAT_ESPERA**.

L'estat **ESTAT_ESPERA** és un estat intermedi que espera un període fix (100 tics) per evitar lectures repetides causades per inestabilitat del joystick. Superat aquest temps, el motor torna a **ESTAT_JOY**. Si s'ha accedit a **ESTAT_LLEGEIX**, s'inicia la recepció d'una cadena de text procedent de la interfície Java. Aquesta cadena es llegeix caràcter a caràcter fins que arriba el final (**\n**), moment en què es passa a **ESTAT_AGAFA_DADES**.

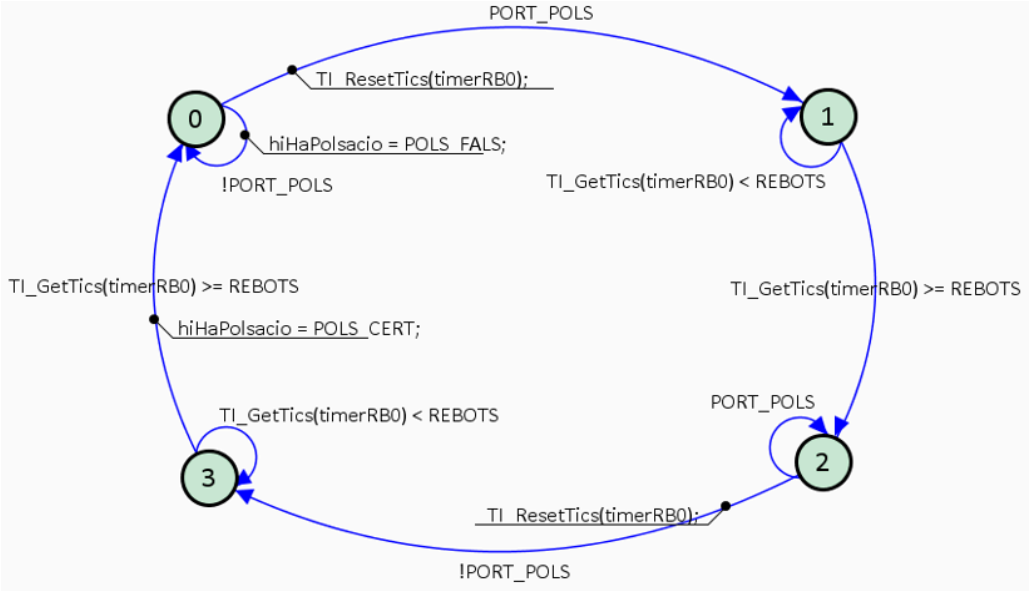
En aquest estat, es compara la cadena rebuda amb les possibles comandes: **INITIALIZE**, **SET_TIME**, **GET_LOGS**, **GET_GRAPH** o **RESET**. En funció del resultat, es canvia l'estat per gestionar la petició. En el cas de **INITIALIZE**, es processen les dades de configuració i es posa el sistema en mode inicialitzat. Si la comanda és **SET_TIME**, es passa a **ESTAT_SET_TIME**, on s'actualitzen les variables d'hora i minut, i posteriorment s'envien al RTC.

Per **GET_LOGS**, el motor entra en **ESTAT_GET_LOGS**, que gestiona l'enviament seqüencial dels logs emmagatzemats a l'EEPROM. Es distingeixen tres fases: preparació del buffer (**CONTROLLER_getLogFormatted()**), enviament caràcter a caràcter i finalització amb el missatge **FINISH**. En **GET_GRAPH**, el motor entra en **ESTAT_GET_GRAPH**, on llegeix les dades de temperatura guardades a la RAM. Aquestes dades es llegeixen i s'envien successivament en el format **DATAGRAPH:XX\r\n**, avançant adreça a adreça fins que s'esgoten, moment en què s'envia el missatge **FINISH**.

En el cas d'un **RESET**, s'inicia una seqüència que passa per **ESTAT_RESET_EEPROM**, **ESTAT_RESET_RAM** i **ESTAT_RESET_SYS**. En cada fase es reinicia un component diferent: primer l'EEPROM, després la RAM i finalment tots els TADs i variables globals. Aquest procés assegura una neteja completa de l'estat del sistema i el retorna a la seva configuració d'origen, sense necessitat de reiniciar el dispositiu.

Finalment, també existeixen els estats **ACTUALITZA_HORA** i **ENVIA_HORA.** El primer llegeix l'hora actual des del RTC i genera el missatge corresponent (**UPDATETIME:HH:MM\r\n**), mentre que el segon l'envia caràcter a caràcter per UART. Un cop acabat, el sistema torna a l'estat **ESTAT_JOY**, preparat per rebre nous inputs. Tot aquest mecanisme fa del **TAD_MENU** un component essencial i sofisticat, capaç de gestionar múltiples funcions sense bloquejar l'execució, mantenint sempre la reactivitat gràcies a una divisió clara d'estats i una gestió acurada del temps i els buffers de comunicació.

MOTOR POLSADOR



El motor cooperatiu del **TAD_POLSADOR** s'encarrega de detectar pulsacions físiques de l'usuari sobre el botó de control, garantint una detecció fiable malgrat els possibles rebots elèctrics típics d'un polsador mecànic. Està dissenyat com una màquina d'estats finits amb quatre fases que permeten gestionar tant la pulsació com la seva estabilització i l'alliberament, tot mantenint la naturalesa no bloquejant pròpia del sistema cooperatiu.

En l'estat 0 (**POLS_NO_PREMUT**), el motor assumeix que el polsador no està actiu. S'hi manté fins que es detecta una transició a nivell alt sobre la línia **PORT_POLS**, indicant una possible pulsació. En aquest cas, es reinicia el temporitzador **timerRB0** i es transita a l'estat 1 (**POLS_REBOTS_IN**), que és el primer estat de desbrossament de rebots.

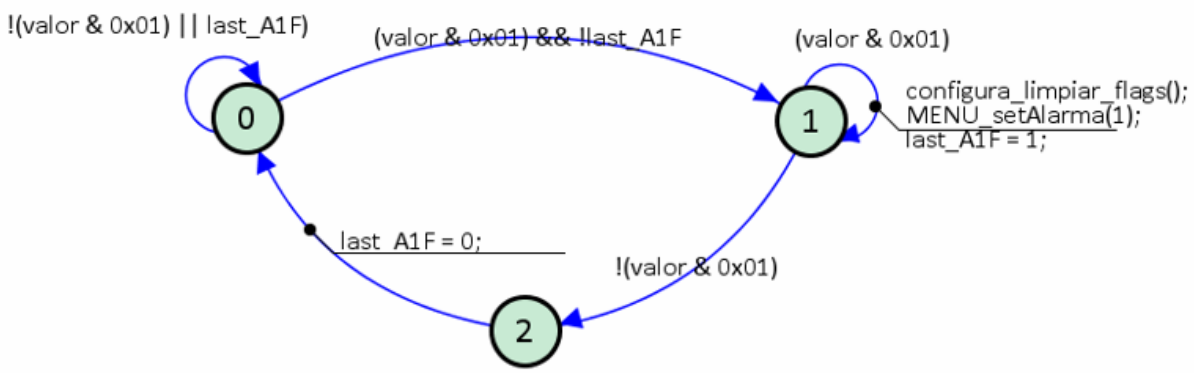
En aquest segon estat, el motor espera un nombre determinat de tics (definit per la constant **REBOTS**) abans de validar que la pulsació és real. Si el temporitzador no ha arribat encara al llindar, el motor roman en aquest estat. Quan el temps establert s'ha complert, es passa a l'estat 2 (**POLS_PREMUT**), on el sistema assumeix que la pulsació ha estat real i estable.

Durant aquest tercer estat, el motor espera que el botó sigui alliberat. Quan la línia **PORT_POLS** passa a nivell baix, es torna a reiniciar el temporitzador i es passa a l'estat 3 (**POLS_REBOTS_OUT**), dedicat a filtrar els possibles rebots d'alliberament. Novament, es manté aquí fins que han transcorregut els tics necessaris. Un cop superat aquest període, es confirma que la pulsació ha acabat correctament i es torna a l'estat inicial.

En aquest moment, es posa a **POL_CERT** el flag hiHaPolsacio, que pot ser consultat des de qualsevol altre mòdul mitjançant la funció **POLSADOR_HiHaPolsacio()**. Aquest flag només es manté actiu durant una iteració i es posa automàticament a **FALS** en l'estat 0 de la següent execució, de manera que cada polsació només pot ser detectada un cop.

Aquest disseny garanteix una detecció robusta, precisa i insensible a les oscil·lacions pròpies del contacte mecànic. A més, el motor opera completament sense bloqueigs, ja que totes les esperes es gestionen mitjançant temporitzadors virtuals i cap estat realitza espera activa. Això assegura una integració fluida amb la resta del sistema, permetent que el polsador pugui ser consultat sense interferències i amb total fiabilitat.

MOTOR RTC



El motor cooperatiu del TAD_RTC és el responsable de detectar quan salta l'alarma del rellotge de temps real DS3231, la qual s'ha configurat prèviament per activar-se exactament cada minut. Aquesta activació es reflecteix en el registre de control del dispositiu mitjançant l'activació del bit A1F del registre 0x0F. El motor s'encarrega de consultar aquest registre de forma cíclica i, si detecta el flag activat, notificar-ho al **TAD_MENU** perquè aquest pugui actualitzar l'hora visible a la interfície.

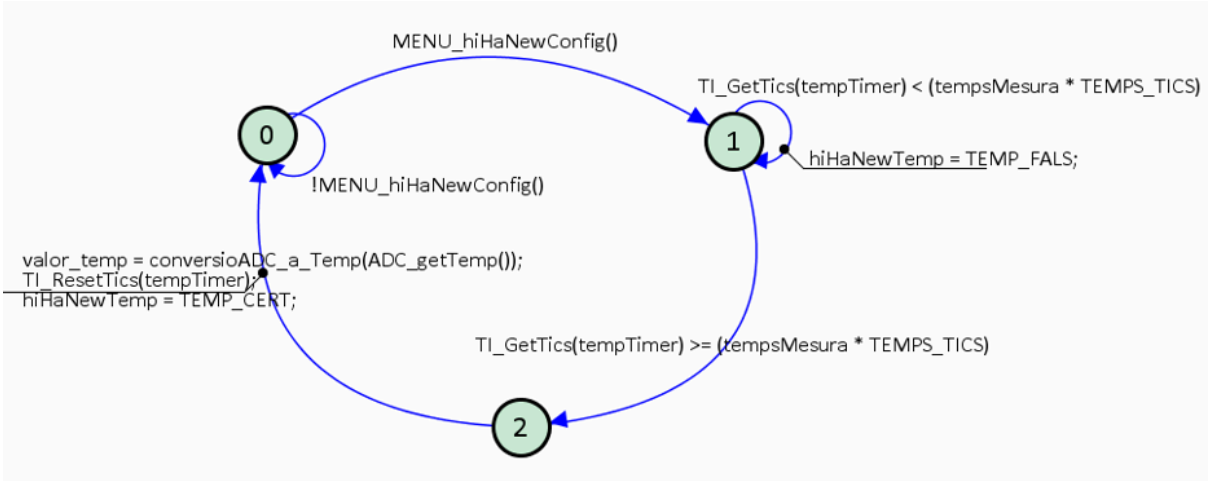
La màquina d'estats del motor està formada per tres estats que s'alternen segons el valor del bit **A1F** i l'estat previ de la variable **last_A1F**. Aquesta variable interna permet detectar la transició de "no activat" a "activat", evitant que es generin múltiples notificacions per una mateixa alarma. En l'estat 0, el motor llegeix el registre **0x0F** mitjançant una operació I²C i extreu el byte valor. Si el bit A1F no està activat (**!(valor & 0x01)**) o si encara està actiu però ja s'ha notificat (**last_A1F == 1**), el motor es manté en aquest estat esperant el següent cicle.

Quan es detecta una transició vàlida, és a dir, quan valor **& 0x0**1 és cert i **last_A1F** és fals, s'entra a l'estat 1. En aquest estat, s'executa la funció **MENU_setAlarma(1)**, la qual notifica al **TAD_MENU** que ha saltat l'alarma i que caldrà actualitzar l'hora a la interfície. Tot seguit, es crida **configura_limpiar_flags()** per netejar el flag **A1F** del registre **0x0F** i es posa **last_A1F = 1** per evitar notificacions duplicades.

Un cop feta la notificació, el motor passa a l'estat 2. En aquest, comprova si el flag **A1F** ha estat netejat (**!(valor & 0x01)**). Si és així, es posa **last_A1F = 0** i es torna a l'estat inicial, preparat per capturar la següent activació de l'alarma. Aquesta transició garanteix que el motor només actuarà un cop per minut, i només quan realment s'ha produït un flanc d'activació.

Tot el procés es fa sense bloquejos, i es basa en lectures puntuals mitjançant el bus I²C sense dependència de cap interrupció. Aquesta estratègia permet integrar la detecció d'alarma dins el model cooperatiu general, garantint que l'hora es manté actualitzada a la interfície amb una latència mínima i sense interferir amb la resta de motors del sistema.

MOTOR TEMPERATURA



El motor cooperatiu del **TAD_TEMP** és l'encarregat de gestionar la lectura periòdica del sensor de temperatura i d'indicar quan hi ha disponible una nova mostra. Aquest motor actua com una màquina d'estats finits formada per tres estats que s'executen seqüencialment segons les condicions temporals configurades i l'estat d'inicialització del sistema.

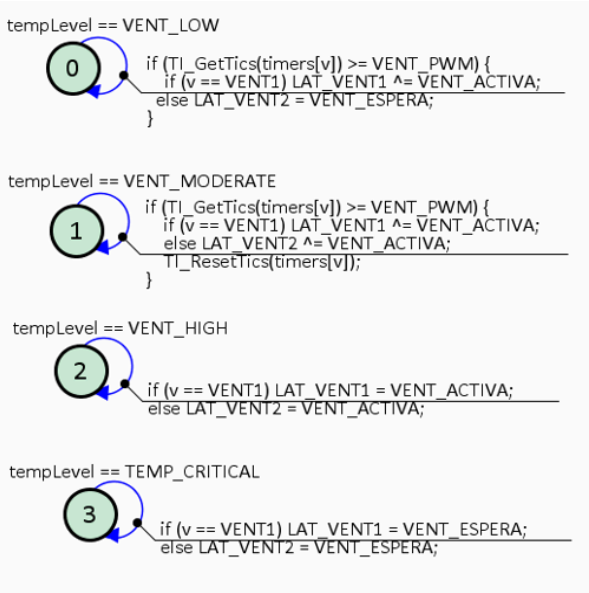
En l'estat 0, el sistema es manté en espera fins que el **TAD_MENU** notifica que el sistema ha estat inicialitzat correctament, mitjançant la crida a **MENU_hiHaNewConfig()**. Mentre aquesta condició no es compleix, el motor no pren cap acció. Tan aviat com es detecta la configuració inicial, es transita a l'estat 1.

En aquest segon estat, el motor inicia el cicle de mesura. Per defecte, es posa el flag **hiHaNewTemp** a **TEMP_FALS,** indicant que no hi ha una nova lectura disponible. A continuació, es manté a l'espera fins que transcorre el temps indicat per la variable **tempsMesura**, mesurat en tics i multiplicat per la constant **TEMPS_TICS** per obtenir una duració temporal ajustada a l'interval desitjat. Aquesta espera es gestiona mitjançant el temporitzador virtual tempTimer, el qual s'inicialitza prèviament a la configuració.

Un cop passat el temps d'espera, el motor accedeix a l'estat 2. En aquest punt, es realitza la lectura directa del sensor de temperatura mitjançant la funció **ADC_getTemp()**. Aquest valor digital és convertit a graus Celsius amb la funció **conversioADC_a_Temp()** i s'emmagatzema a la variable **valor_temp**. Tot seguit, es posa a **TEMP_CERT** el flag **hiHaNewTemp** per indicar que hi ha una nova lectura disponible per la resta de mòduls, com ara el **TAD_CONTROLLER** o el **TAD_LED**. Finalment, es reinicia el temporitzador i es retorna a l'estat d'espera per iniciar un nou cicle.

Aquest motor opera de manera no bloquejant i sincronitzada amb la resta del sistema. No utilitza interrupcions ni operacions de retard actiu, sinó temporitzadors virtuals que permeten mesurar intervals de temps de forma eficient. Això garanteix que el sistema pugui continuar processant altres motors mentre el **TAD_TEMP** espera el moment òptim per fer una nova lectura. El seu disseny permet ajustar fàcilment la freqüència de mesura des del menú, fent-lo un component flexible, lleuger i coherent amb l'arquitectura cooperativa del projecte.

MOTOR VENTILADOR



El motor del **TAD_VENTILADOR** s'encarrega de controlar l'estat de funcionament dels dos ventiladors del sistema segons el nivell de temperatura detectat pel **TAD_TEMP**. Aquest motor és cridat per separat per a cada ventilador, identificat pel paràmetre v (sent **VENT1** o **VENT2**), i es basa en un model de quatre estats que representen els diferents nivells de resposta tèrmica: **VENT_LOW**, **VENT_MODERATE**, **VENT_HIGH** i **TEMP_CRITICAL**.

Quan el nivell de temperatura és baix (**VENT_LOW**), només el primer ventilador (**VENT1**) opera a velocitat moderada mitjançant PWM. El motor consulta si han passat els tics especificats per **VENT_PWM** mitjançant el temporitzador timers[v] i, si és així, alterna l'estat del pin corresponent (**toggle**) per generar una ona quadrada. El segon ventilador es manté apagat en aquest cas.

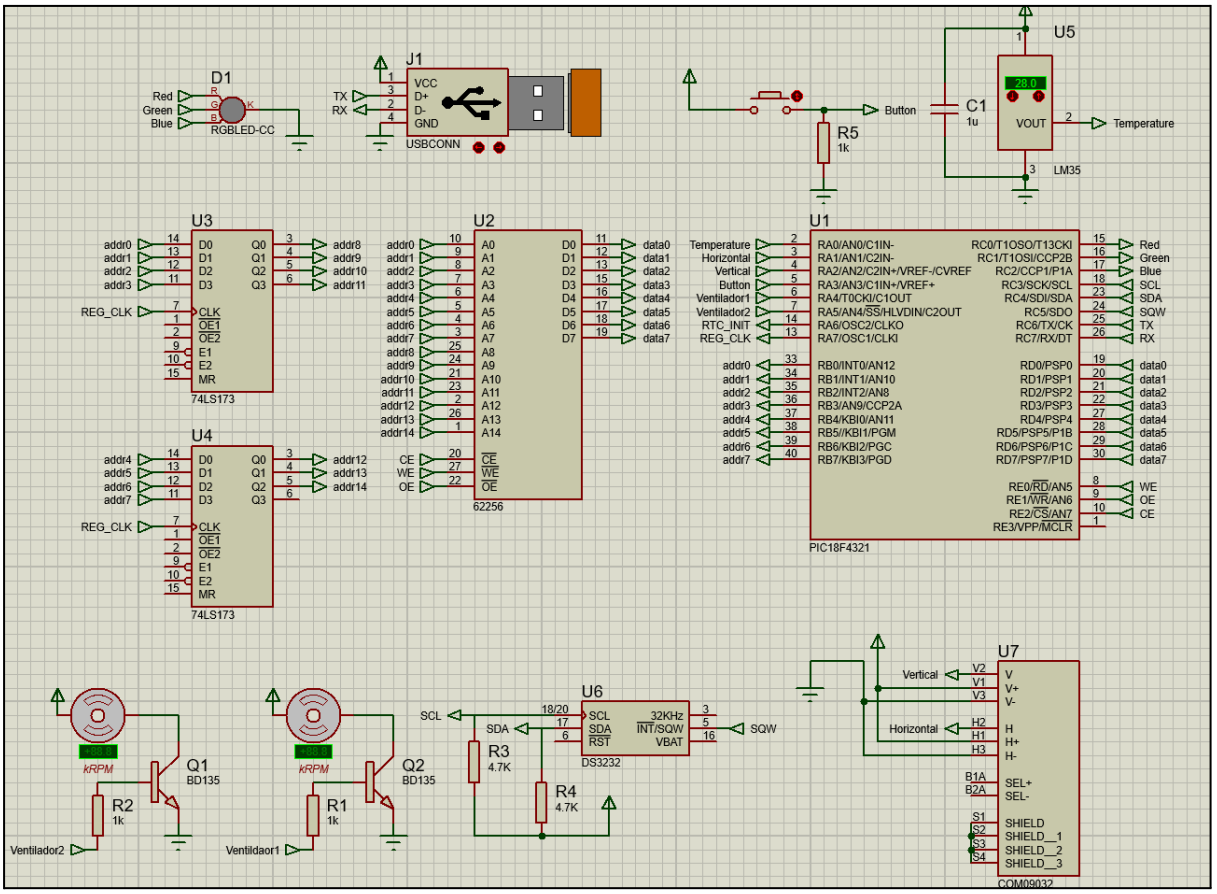
En el nivell **VENT_MODERATE**, ambdós ventiladors funcionen a velocitat moderada, també utilitzant PWM. El motor alterna l'estat de cadascun independentment, sempre que hagi transcorregut el temps de **VENT_PWM**, i després reinicia el temporitzador. Aquesta estratègia produeix un senyal de cicle útil 50%, generat completament via software, per controlar els transistors de potència que activen els motors físics.

Quan s'assoleix el nivell **VENT_HIGH**, es busca la màxima refrigeració i els ventiladors es deixen activats permanentment. En aquest cas, el motor estableix directament l'estat **VENT_ACTIVA** al pin corresponent, eliminant qualsevol modulació. Això assegura el màxim flux d'aire per dissipar calor.

Finalment, en estat **TEMP_CRITICAL**, la resposta del sistema canvia radicalment per protegir el maquinari. Tant **VENT1** com **VENT2** s'apaguen completament, fixant les línies de control a **VENT_ESPERA**. Aquesta decisió es pren per evitar danys als components en situacions extremes o errònies, com fallades de sensors o situacions ambientals perilloses.

El motor opera amb total compatibilitat amb el model cooperatiu. Cada execució només realitza comprovacions i accions simples sobre el GPIO, i utilitza temporitzadors virtuals per temporitzar el PWM, evitant qualsevol retard actiu. A més, la independència del motor per a cada ventilador permet modular el comportament de manera flexible i optimitzada, garantint estabilitat, eficiència energètica i una bona integració amb la resta de l'arquitectura.

Esquema elèctric



L'esquema elèctric del sistema s'ha dissenyat per reflectir de manera clara i funcional l'arquitectura modular implementada al programari. Al centre de la topologia es troba el microcontrolador PIC18F4321, encarregat de coordinar la lectura de sensors, el control d'actuadors, i la comunicació amb dispositius externs. Cada perifèric està connectat d'acord amb les restriccions pròpies del maquinari i tenint en compte criteris d'estabilitat, simplicitat i seguretat.

A la part superior dreta de l'esquema, s'hi troba el sensor de temperatura LM35, la sortida analògica del qual es connecta a l'entrada **RA0/AN0** del microcontrolador. Aquest senyal és digitalitzat pel mòdul ADC per obtenir la temperatura ambiental. A sota, s'hi connecta un pulsador vinculat a **RA3**. Aquesta entrada és utilitzada pel sistema per detectar seleccions de menú o confirmacions de l'usuari.

Els dos ventiladors s'alimenten mitjançant transistors NPN (BD135), els quals són activats directament pels pins **RA4** i **RA**5. Aquest mètode permet al microcontrolador controlar la potència aplicada als ventiladors mitjançant PWM software, generat a partir del motor cooperatiu corresponent. Cada transistor inclou una resistència base per limitar el corrent i protegir el dispositiu.

Pel que fa a la RAM externa, s'utilitza un xip 62256 de 32 KB organitzat amb un bus de dades de 8 bits i un bus d'adreces de 15 bits. La part baixa de les adreces (**A0-A7**) es connecta directament al port B del microcontrolador, mentre que les adreces altes (**A8-A1**4) es gestionen mitjançant dos registres de desplaçament 74LS173, controlats amb un senyal de rellotge comú (**REG_CLK**) vinculat al pin **RA7**. Aquest enfocament permet estalviar línies GPIO, ja que només cal escriure les adreces altes al port de dades i fer un pols de rellotge per actualitzar-les.

Els senyals de control de la RAM (OE, WE i CE) es connecten respectivament als pins **RE1**, **RE0** i **RE2**, definits com sortides digitals. El bus de dades de la RAM està vinculat al port D i és gestionat de manera bidireccional mitjançant la configuració del registre **TRISD**, controlant si el port actua com a entrada o sortida segons si es fa una lectura o una escriptura.

A nivell de comunicació externa, el sistema incorpora un mòdul USB-to-Serial (vist com un connector USB a l'esquema), els pins del qual es connecten a TX i RX del microcontrolador (**RC6 i RC7**). Aquesta interfície UART permet enviar i rebre dades amb una aplicació Java en el PC, que actua com a interfície d'usuari del sistema.

El rellotge de temps real DS3232 es connecta al microcontrolador mitjançant el bus I²C. Els pins SCL i SDA es connecten a **RC3** i **RC4**, amb resistències de pull-up de 4.7 kΩ. A més, la sortida de senyal de 1 Hz (SQW/INT) del RTC es connecta a **RA5**, que és llegida periòdicament pel **TAD_RTC** per detectar canvis de minut. El mòdul RTC també inclou una bateria de reserva que assegura el manteniment de l'hora quan el sistema està desconnectat.

També es connecta un joystick analògic amb dues sortides, corresponents als eixos X i Y, que s'envien a les entrades **RA1** i **RA2** del microcontrolador. Aquestes són llegides per l'ADC i interpretades pel **TAD_JOYSTICK** per navegar pel menú. El **LED RGB** està connectat als pins **RC0**, **RC1** i **RC2**, els quals són gestionats mitjançant el **TAD_LED** per reflectir l'estat tèrmic del sistema. Finalment, el pin **RA6**, a més de controlar el rellotge dels registres d'adreça, s'utilitza també com a indicador visual del correcte funcionament del RTC, activant o apagant un LED en funció del resultat de la inicialització.

Aquesta distribució física de components permet una clara separació funcional, facilita la depuració i assegura que cada mòdul programari (TAD) disposa del maquinari necessari per operar de forma estable. El conjunt resulta en un sistema integrat i eficient, tant a nivell de recursos com de disposició sobre placa.

Problemes observats

Durant el desenvolupament i validació del sistema s'han identificat diversos problemes que han requerit una anàlisi acurada i ajustos en el codi o el maquinari per poder garantir el correcte funcionament global del projecte. El més destacat ha estat la dificultat inicial per establir una comunicació fiable amb el rellotge de temps real mitjançant el bus I²C. En les primeres proves, el sistema no era capaç de llegir ni escriure correctament als registres del DS3231, fet que impedia configurar l'hora i detectar el salt d'alarma. El problema s'ha acabat resolent revisant la seqüència d'inicialització, ajustant la configuració dels registres de control i afegint comprovacions d'error després de cada operació per detectar possibles fallades de transmissió.

Un altre repte rellevant ha estat la implementació de l'opció de reinicialització del sistema des de la interfície Java. Aquesta funcionalitat requeria esborrar completament la informació emmagatzemada a l'EEPROM i la RAM, reiniciar els perifèrics i deixar totes les variables globals en un estat coherent. Ha estat necessari introduir una seqüència d'estats específica dins del **TAD_MENU** per gestionar aquest procés de manera ordenada i sense bloquejar l'execució de la resta del sistema. També s'ha hagut de garantir que la reinicialització fos reversible i segura, fins i tot si es produïa durant una operació activa.

A nivell d'interfície, un dels problemes més subtils ha estat relacionat amb el control de les fletxes que permeten modificar l'hora des de la interfície gràfica Java. Tot i que l'aplicació mostrava correctament les hores modificades mitjançant aquest sistema, el missatge resultant de la comanda **SET_TIME** no sempre coincidia amb l'hora seleccionada visualment, generant desajustos que dificultaven la comprovació del funcionament del RTC. Aquest problema va provocar confusió durant la fase de proves i va requerir una verificació manual del contingut exacte de les cadenes rebudes per UART, així com una revisió acurada de l'estat intern del **TAD_MENU** abans de permetre la configuració final de l'hora.

També s'ha detectat en alguns casos un comportament inesperat del LED RGB durant la transició entre modes de parpelleig i colors fixes. Això es devia a una solapació d'instruccions entre el motor del **TAD_LED** i l'estat del sistema, que no sempre deixava clar si el sistema havia d'entrar en mode crític o mantenir una indicació estàtica. El problema s'ha resolt reestructurant els estats del motor i separant clarament la gestió del parpelleig de la lògica de color.

Finalment, cal destacar que una part significativa del desenvolupament s'ha centrat a garantir que tots els motors cooperatius siguin realment no bloquejants i compatibles entre si. Alguns estats intermedis, especialment en els mòduls que treballen amb la UART o amb memòries, han requerit refactorització per evitar que una lectura incompleta o un temps d'espera massa llarg provoqués una pèrdua de reactivitat general del sistema. Aquesta depuració ha estat progressiva, però essencial per assegurar la fluïdesa i robustesa de tot el conjunt.

Conclusions

El desenvolupament d'aquest sistema cooperatiu embegut ha representat un repte tècnic considerable, tant pel nombre d'elements implicats com per la necessitat d'una coordinació fluida i no bloquejant entre ells. A través d'una arquitectura basada en motors cooperatius, hem aconseguit gestionar múltiples perifèrics (sensor analògic, memòries, LED RGB, joystick, ventiladors, RTC i comunicació UART) garantint la seva compatibilitat temporal dins d'un mateix bucle principal. Aquesta metodologia ha permès mantenir una estructura clara, modular i altament escalable, facilitant el procés de depuració i ampliació de funcionalitats al llarg del projecte.

La implementació de TADs específics per cada funcionalitat ha resultat una decisió encertada, ja que ha contribuït a encapsular el comportament de cada perifèric i a definir una interfície clara entre mòduls. Aquesta modularitat ha estat especialment útil durant la integració de les memòries (RAM i EEPROM) i del rellotge RTC, així com en la gestió de la configuració inicial a través de la interfície Java. En aquest context, el **TAD_MENU** ha actuat com a centre de control del sistema, coordinant la comunicació amb l'usuari, la interpretació de comandes i la transició entre estats interns.

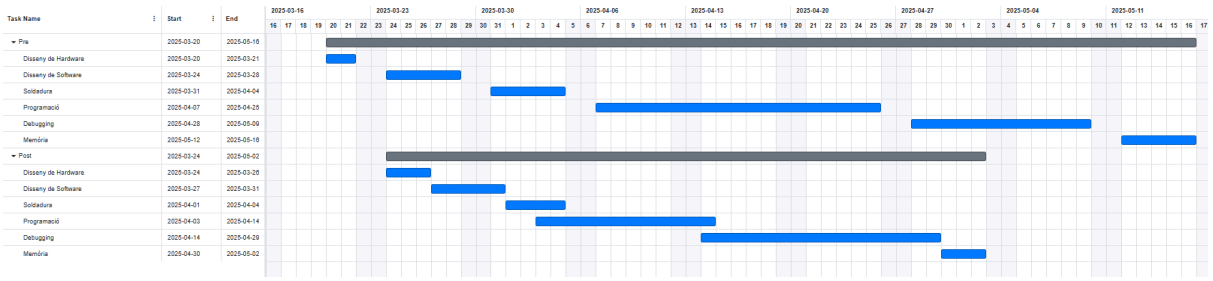
S'ha validat que l'ús de temporitzadors virtuals gestionats per una única interrupció del **TMR0** (cada 2 ms) és suficient per controlar el temps d'espera en tots els motors, amb una resolució adequada per a la majoria d'operacions. Aquest mecanisme ha permès evitar l'ús d'interrupcions múltiples o perifèrics avançats, fet que simplifica el disseny i facilita el manteniment del codi. La precisió aconseguida en la temporització, especialment per a la lectura de sensors i la generació de PWM software, ha estat suficient per complir els requisits funcionals del sistema.

A nivell funcional, s'ha pogut implementar amb èxit la lectura de temperatura, el control adaptatiu dels ventiladors, la gestió d'un LED RGB en funció de l'estat tèrmic, l'enregistrament de dades a memòria RAM i EEPROM, la visualització en gràfic mitjançant Java i la sincronització horària a través d'un RTC. L'esquema elèctric resultant reflecteix fidelment aquesta arquitectura, amb una clara assignació de pins i una topologia que minimitza el nombre de línies necessàries mitjançant l'ús de registres de desplaçament per controlar la memòria.

Tot i els problemes inicials relacionats amb la comunicació I²C, la interpretació de missatges des de la interfície Java i la gestió d'alguns estats complexos, s'ha aconseguit construir un sistema estable i fiable. L'experiència adquirida durant la resolució d'aquests problemes ha estat especialment valuosa per entendre la importància de la sincronització, la verificació de condicions en motors cooperatius i la necessitat de dissenyar estats robustos i ben definits.

En conclusió, aquest projecte ha permès posar en pràctica coneixements de programació estructurada, electrònica digital, gestió del temps i integració de perifèrics dins d'un sistema embegut cooperatiu. El resultat final és un sistema funcional, clarament estructurat i preparat per a ser millorat o adaptat a futures extensions, amb una base tècnica sòlida i una arquitectura eficient que reflecteix una comprensió profunda tant del maquinari com del programari implicats.

Planificació



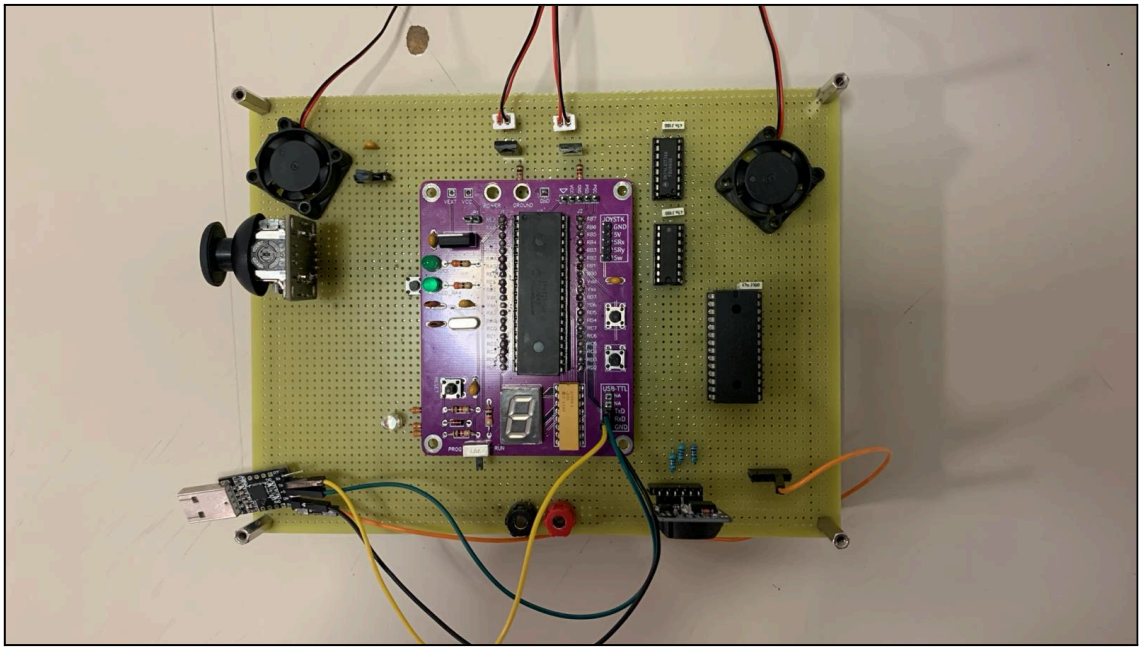
La planificació inicial del projecte es va dividir en sis fases diferenciades: disseny de hardware, disseny de software, soldadura, programació, debugging i redacció de la memòria. Aquesta estructura es va definir abans d'iniciar la pràctica i es reflecteix a la part superior del diagrama de Gantt, amb una franja grisa que abasta tot el període previst d'execució. Inicialment es contemplava una durada aproximada de vuit setmanes per completar totes les tasques.

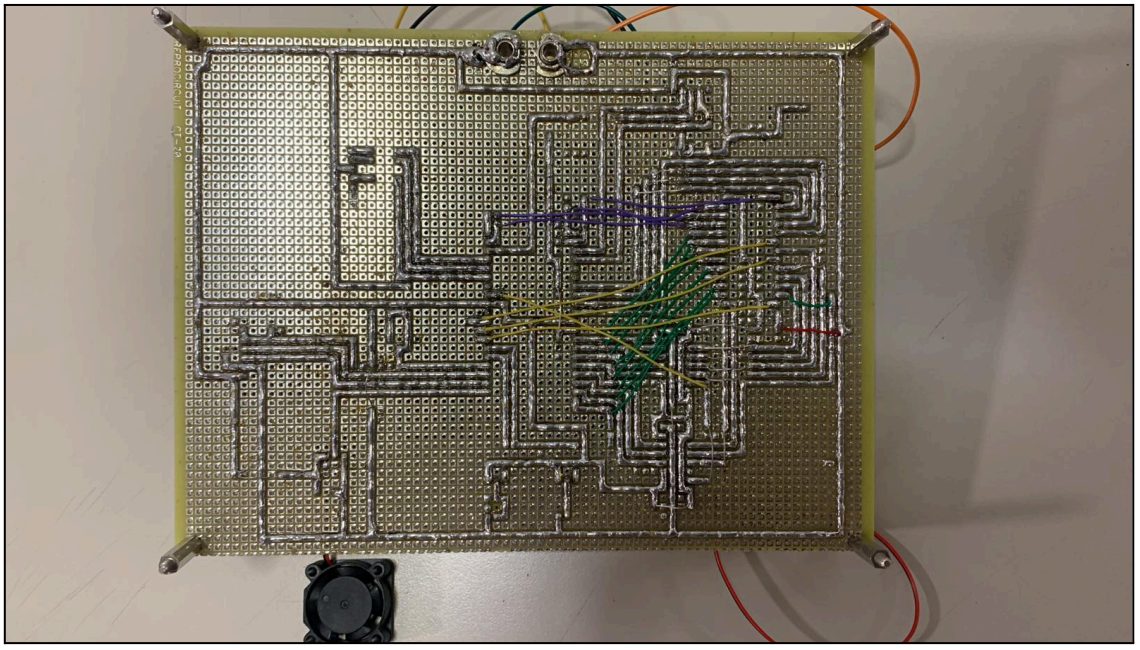
Tanmateix, la pràctica es va començar amb uns dies de retard respecte a la planificació prevista. Tot i això, gràcies a una dedicació més intensiva del temps disponible, incloent hores extres entre setmana i caps de setmana, es va aconseguir avançar més ràpidament del previst en cada fase.

A la part inferior del diagrama es mostra el desenvolupament real, amb una segona franja grisa que representa la durada total final, més curta que la inicialment prevista (5-6 setmanes). Les fases blaves mostren l'execució efectiva de cada tasca, i es pot veure com la seva distribució temporal és més concentrada, amb menys espai entre fases i algunes que s'han solapat parcialment.

Aquest desfasament positiu ha permès acabar la pràctica abans del termini previst, deixant més temps per fer una revisió global i optimitzar aspectes com la memòria, la modularitat del codi i la documentació.

Fotografies i vídeo demostració





Video: https://drive.google.com/file/d/10RCqmrh2PLaP4j1QqVR-ZPa_9QMwfn-i/view?usp=sharing