

UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
MAESTRÍA EN SISTEMAS EMBEBIDOS



MEMORIA DEL TRABAJO FINAL

**Sistema de control para estación
autónoma marítima de monitoreo de
ruido ambiente**

Autor:
Esp. Ing. Patricio Bos

Director:
Dr. Ing. Ariel Lutenberg (FIUBA)

Jurados:
Dr. Ing Pablo Gómez (FIUBA)
Ing. Juan Manuel Cruz (FIUBA, UTN-FRBA)
Mg. Lic. Igor Prario (FIUBA)

*Este trabajo fue realizado en las Ciudad Autónoma de Buenos Aires, entre enero
de 2017 y diciembre de 2018.*

Resumen

La presente memoria describe el diseño e implementación de un sistema embebido para el control de una estación de monitoreo de ruido ambiente submarino. El conocimiento del nivel de ruido submarino es importante para distintas disciplinas dentro de la oceanografía acústica, estudios de impacto ambiental y diversas aplicaciones que utilicen sistemas SONAR dentro del ámbito científico, civil y militar, entre otras.

En este documento se describe la implementación de un *firmware* multicore modular, junto con los mecanismos de comunicación y sincronización entre procesadores implementados. Asimismo, se incluye la ingeniería de detalle de los módulos desarrollados y la documentación de *testing* junto con información para la trazabilidad de los requerimientos en las distintas etapas del proyecto.

Para la realización de este trabajo se adoptó un modelo de desarrollo basado en herramientas de control de versiones y se utilizaron técnicas de programación multi-tarea cooperativa.

Agradecimientos

Me gustaría agradecer en estas líneas la ayuda que las personas y colegas me han prestado durante el proceso de elaboración y redacción de este trabajo.

En primer lugar, quisiera agradecer a Carla, mi compañera, por su cariñoso e incondicional acompañamiento como sólo ella sabe ofrecer, sin el cual esta empresa no hubiera sido siquiera imaginable.

A mi director, el Dr. Ing. Ariel Lutenberg por la orientación y sus valiosos aportes en la redacción de esta memoria; a él junto con el Dr. Ing. Pablo Gómez por su compañerismo en la tarea docente, fundamental para la culminación de este trabajo en fecha. A ellos dos también, en su carácter de autoridades de la Maestría por haber generado y sostener cotidianamente un ámbito de formación profesional de excelencia con un fuerte sentido de pertenencia.

A mis compañeros de la División Acústica Submarina con quienes comparto amablemente jornadas laborales llenas de desafíos y en particular a la Lic. Silvia Blanc por promover desinteresadamente mi crecimiento humano y profesional.

A todas las personas que cuidaron a Dante, familiares y amigos, en mis largas jornadas de trabajo.

A todos ellos, mi más sentido agradecimiento.

Índice general

Resumen	III
1. Introducción General	1
1.1. Descripción técnica-conceptual del proyecto	1
1.1.1. Ruido ambiente submarino	2
1.2. Motivación	3
1.3. Objetivos y alcance	3
2. Introducción Específica	5
2.1. Requerimientos	5
2.2. Planificación	6
2.3. Metodología	8
2.3.1. Control de versiones	8
2.3.2. Paradigma de desarrollo basado en pruebas	10
2.3.3. Programación concurrente con Protothreads	11
2.4. Arquitectura multicore	12
2.4.1. Inter Processor Communications	13
Interrupciones cruzadas	13
Colas de mensajes	14
3. Diseño e Implementación	17
3.1. Diseño de módulos y definición de interfaces	17
3.2. Módulo de almacenamiento	20
3.2.1. Medio físico	20
3.2.2. Máquina de Estados Finitos	21
3.3. Módulo de adquisición	23
3.3.1. Sensor de temperatura	23
3.3.2. Máquina de Estados Finitos	25
3.4. Módulo interfaz máquina-hombre	26
3.4.1. Interfaz	27
3.4.2. Máquina de estados finitos	32
3.5. Módulo de control	34
4. Ensayos y Resultados	39
4.1. Test Master Plan	39
4.1.1. Pruebas unitarias	39
Módulo Almacenamiento	40
Módulo adquisición	41
Módulo HMI	42
Módulo Control	43
4.1.2. Pruebas funcionales	44
4.1.3. Pruebas de sistema	47

5. Conclusiones	53
5.1. Conclusiones generales	53
5.1.1. Técnicas útiles	54
5.1.2. Metas alcanzadas	55
5.2. Próximos pasos	56
A. Plantilla para casos de uso	57
Bibliografía	59

Índice de figuras

1.1. Diagrama en bloques general del sistema.	3
1.2. Diagrama en bloques implementado	4
2.1. Diagrama <i>Activity on Node</i>	7
2.2. Esquema del flujo de trabajo entre repositorios	8
2.3. Modelo de ramas utilizado en git	9
2.4. Esquema de comunicación entre procesadores	14
3.1. Estructura de capas para el <i>firmware</i>	17
3.2. Diseño de MEF genérica para los módulos	19
3.3. (A) Diagrama de conexionado eléctrico y (B) Lector de tarjetas SD utilizado.	20
3.4. Diagrama de capas de fatFs	21
3.5. MEF principal del módulo de almacenamiento sdCard	22
3.6. Diagrama en bloques del sensor de temperatura DS18B20	24
3.7.	24
3.8. MEF principal del módulo de adquisición de temperatura	25
3.11. Pantalla de configuración del módulo de adquisición.	30
3.12. Vista de la pantalla principal del modo debug de la interfaz máquina-hombre.	30
3.13. MEF principal del módulo de HMI	33
4.1. Casos de prueba para <i>test</i> unitarios del módulo de almacenamiento	40
4.2. Casos de prueba para <i>test</i> unitarios del módulo de adquisición . . .	41
4.3. Casos de prueba para <i>test</i> unitarios del módulo de interfaz máquina-hombre	42
4.4. Casos de prueba para <i>test</i> unitarios del módulo de control	43
4.5. Salida por consola del test funcional para el módulo de almacenamiento	44
4.6. Planilla de caso de uso UC01. Medición autónoma de temperatura en forma periódica. Se indica en color verde la post condición de éxito del ensayo.	48
4.7. Planilla de casos de uso UC02. Cambio de período de adquisición de valores de temperatura. Se indica en color verde la post condición de éxito del ensayo.	49
4.8. Planilla de casos de uso UC03. Cambio de perfil de consumo de energía. Se indica en color verde la post condición de éxito del ensayo.	50
4.9. Banco de mediciones para los ensayos de sistema	51

Índice de Tablas

2.1. Etapas principales del proyecto	6
2.2. Asignación de bloques de memoria para el Cortex-M4	12
2.3. Asignación de bloques de memoria para el Cortex-M0	12
3.1. Matriz de trazabilidad de requerimientos funcionales	18
3.2. Alternativas de medios físicos evaluados.	20
3.3. Descripción de los estados de la MEF principal del módulo de almacenamiento.	22
3.4. Alternativas de sensor de temperatura evaluadas.	23
3.5. Descripción de los estados de la MEF principal del módulo de adquisición.	26
3.6. Opciones disponibles en el modo configuración de la interfaz. Se indica el comando para seleccionar la opción y si ésta se encuentra implementada o maquetada.	31
3.7. Opciones disponibles en el modo debug de la interfaz. Se indica el comando para seleccionar la opción y si ésta se encuentra implementada o maquetada.	32
3.8. Descripción de los estados de la MEF principal del módulo HMI. .	33
3.9. Asignación de módulos a los procesadores	38
4.1. Matriz de trazabilidad de requerimientos con test de casos de uso. .	51
5.1. Requerimientos alcanzados.	55

Para Dante y Facundo.

Capítulo 1

Introducción General

En este capítulo se introduce el campo de la acústica submarina y la importancia del parámetro SONAR Ruido Ambiente Submarino como motivación para la realización de este trabajo. Asimismo, se presentan los objetivos y el alcance del proyecto.

1.1. Descripción técnica-conceptual del proyecto

La acústica submarina estudia la propagación del sonido en el agua y la interacción de las ondas mecánicas que constituyen el sonido con el agua, los elementos dispersores presentes y las interfaces aire-agua y agua-lecho marino. Debido a que sufre menor atenuación que otras formas de radiación, el sonido es ampliamente empleado por el hombre en su exploración de los océanos. Las frecuencias típicas utilizadas se encuentran en el rango comprendido entre ~ 10 Hz y 1 MHz, dependiendo de la aplicación.

Los sistemas que utilizan la propagación del sonido bajo el agua con diversos fines se conocen como sistemas SONAR (del inglés, *SOund Navigation And Ranging*).

La propagación del sonido en el agua depende de diversos factores. La dirección de propagación está determinada principalmente por el gradiente vertical de velocidades del sonido, que a su vez depende fundamentalmente de la temperatura y la salinidad del agua. El perfil de velocidades del sonido puede causar zonas de baja intensidad del sonido, llamadas “zonas de sombra”, y regiones de alta intensidad llamadas “cáusticas”. Estas zonas pueden hallarse con el método de trazado de rayos [1].

El sonido en el agua puede propagarse a grandes distancias, en el orden de miles de kilómetros, debido a la presencia de un canal especial que actúa como guía de onda para el sonido, conocido como SOFAR (**S**Ound **F**ixing **A**nd **R**anging) que se produce, bajo ciertas condiciones, a la profundidad donde el gradiente de velocidades del sonido alcanza un mínimo [2].

Los distintos fenómenos que afectan al sonido submarino pueden ser convenientemente agrupados en un pequeño número de parámetros conocidos como parámetros SONAR que se pueden relacionar entre sí mediante las ecuaciones SONAR [3]. Estas ecuaciones exhiben las relaciones de trabajo que agrupan los efectos del medio de propagación, el blanco y el equipamiento utilizado y constituyen las herramientas básicas para los profesionales que trabajen en aplicaciones de acústica submarina.

1.1.1. Ruido ambiente submarino

En el campo de la acústica submarina resulta muy relevante el conocimiento del parámetro SONAR Nivel de Ruido en el mar (NL: *Noise Level*), que incluye el Ruido Ambiente propiamente dicho (NLa) y el Ruido Propio (NLp) asociado al sistema de medición. En el caso de escucha pasiva (estudios de impacto ambiental sobre mamíferos marinos a bajas frecuencias o detección subacuática efectuada desde vehículos submarinos), el NL está dominado por el NLa.

Conceptualmente, el Nivel de Ruido en el mar está asociado al ruido de “fondo” (*background*) remanente en ausencia de toda otra fuente identificable. Es el nivel de energía acústica mínimo que debe tener una señal para ser detectada.

Cabe destacar que el ruido subacuático puede clasificarse esencialmente en tres tipos:

- Ambiente: comúnmente denominado ruido de fondo, se mide omnidireccionalmente y es originado principalmente por ruido en la superficie marina (debido al viento, oleaje o lluvia), ruido de origen biológico (producido por peces, mamíferos e invertebrados), ruido sísmico o geoacústico natural, ruido de tráfico marítimo (originado por tráfico marítimo distante).
- Ruido Radiado: originado por una fuente específica tal como un buque en particular, plataformas de explotación de petróleo o gas, instalaciones de exploración y perforación, instalaciones de generación eléctrica, etc.
- Ruido Propio: generado por el propio sistema de electrónico de medición de ruido y por la plataforma donde se encuentre instalado.

El objetivo de este proyecto consiste en diseñar e implementar un sistema embebido para controlar una estación autónoma para la medición in-situ del ruido ambiente submarino para ser instalada en regiones de interés en el mar argentino y con la capacidad de transmitir datos a una estación receptora en tierra. Este desarrollo permitirá disponer de series temporales de ruido ambiente submarino durante períodos lo suficientemente largos como para analizar los resultados mediante modelos teóricos y/o empíricos que contribuyan a incrementar el conocimiento de dicho parámetro, especialmente a nivel local.

Se presenta un diagrama en bloques del sistema en la figura 1.1 donde se pueden observar los distintos módulos que componen el sistema. En color verde los componentes asociados a la gestión de energía; en celeste los componentes asociados al almacenamiento de datos; en amarillo los componentes asociados a la gestión de las comunicaciones; en color azul los bloques del subsistema de adquisición de datos y finalmente, en color naranja se destaca el módulo central de procesamiento.

Para alcanzar el objetivo general se dispone de un equipo de trabajo multidisciplinario con conocimientos teóricos de los fenómenos físicos subyacentes a la propagación del sonido en el medio submarino, acceso a bibliografía especializada en acústica submarina y conocimientos de ingeniería en el campo de los sistemas embebidos.

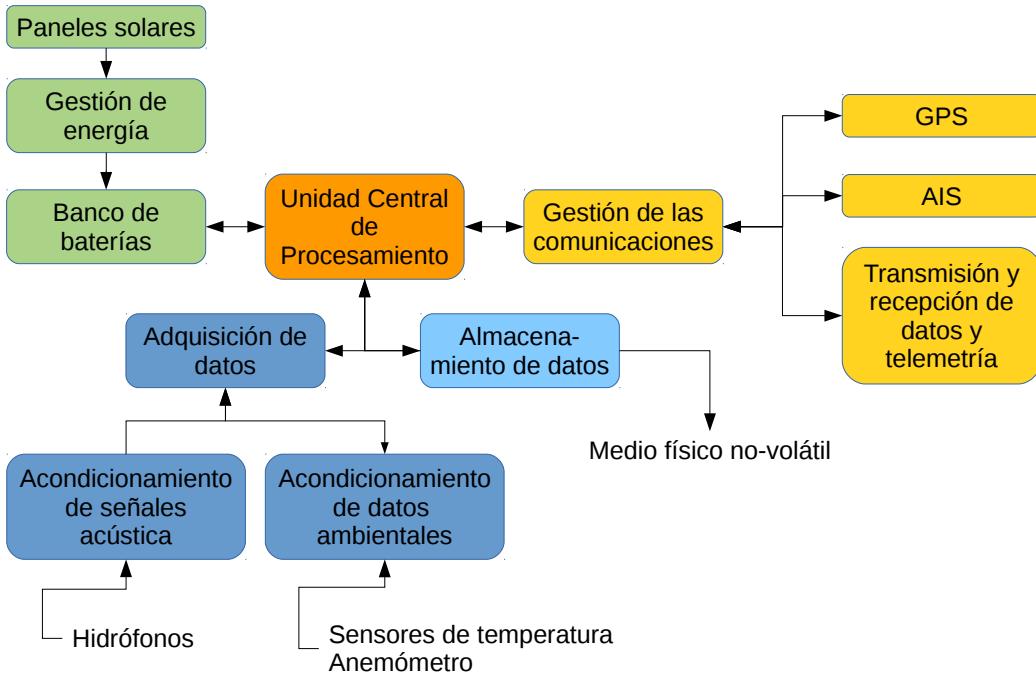


FIGURA 1.1: Diagrama en bloques general. Se diferencian por color los sub-módulos funcionales: energía; unidad central de procesamiento; comunicaciones; adquisición y almacenamiento.

1.2. Motivación

El conocimiento de valores de NL es fundamental en aplicaciones tales como oceanografía acústica, predicción SONAR, exploración geofísica, comunicación subacuática e ingeniería offshore, entre otras.

Por otra parte, existen muy pocas normas a nivel internacional para la estandarización de la medición in-situ del ruido ambiente subacuático. Si bien en acústica aérea sí existen estándares nacionales e internacionales muy aceptados, éstos no pueden extrapolarse fácilmente a la acústica subacuática dadas las diferentes características físicas del fluido en el cual se propaga el sonido, respectivamente.

Actualmente, en el ámbito de la comunidad científica internacional, existe una creciente necesidad de medición y monitoreo del ruido subacuático. El interés está parcialmente motivado por un marco regulatorio internacional en lo concerniente al impacto ambiental del ruido subacuático de origen antrópico y principalmente para la evaluación de los efectos sobre la vida marina.

1.3. Objetivos y alcance

En particular, para el trabajo final de la Maestría en Sistemas Embebidos, se realizará una primera iteración sobre el ciclo de diseño centrada en la programación del sistema embebido que constituye la unidad central de procesamiento de la boyta. Se propone desarrollar sobre la plataforma CIAA-NXP, un firmware multicore de control que utilice ambos procesadores del microcontrolador LPC4337 y sea capaz de cumplir las siguientes funciones:

- Adquirir datos ambientales de temperatura y velocidad de viento.
- Controlar el sistema mediante una interfaz serie.
- Almacenar los datos en una memoria no volátil.

En la primera iteración, se contempla la posibilidad de simular algún elemento del sistema según sea necesario para avanzar rápidamente en el diseño del firmware de control y las funciones mencionadas.

A los fines prácticos de cumplir los requerimientos de tiempo del trabajo final de maestría, quedarán excluidos del diseño:

- La transmisión de datos en tiempo real a una estación receptora en tierra.
- Consideraciones mecánicas del proyecto.
- La gestión de energía.
- La gestión y control del señalamiento reglamentario marítimo.
- La adquisición de señales acústicas.

Según un estudio preliminar, para el registro de señales acústicas resulta necesario una placa de adquisición A/D con características muy específicas en cuanto a frecuencia de muestreo, bits de resolución y figura de ruido, del tipo NI USB-6356 [4] o equivalente. Este tipo de placas poseen *drivers* propietarios cerrados que, en principio, no fue posible utilizar con la CIAA-NXP. Por este motivo, la adquisición de señales acústicas también queda excluida del alcance en esta primera iteración.

Se muestra en la figura 1.2 un diagrama en bloques reducido con los componentes del sistema incluídos en la presente memoria.

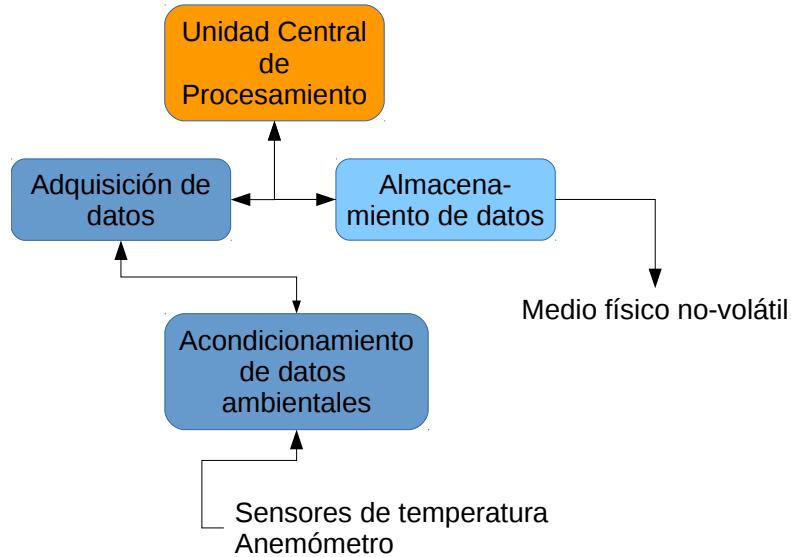


FIGURA 1.2: Diagrama en bloques del sistema implementado. Se diferencian por color los distintos sub-módulos funcionales incluidos en el alcance del proyecto.

Capítulo 2

Introducción Específica

En este capítulo se identifican aspectos relevantes de la planificación. Asimismo, se describen las herramientas empleadas para la realización de este trabajo.

2.1. Requerimientos

A continuación se enumeran los requerimientos del proyecto:

1. Requerimientos de documentación:
 - 1.1 Se debe generar un Memoria Técnica con la documentación de ingeniería detallada.
 - 1.2 Se debe generar un documento de casos de prueba.
2. Requerimientos funcionales del sistema:
 - 2.1 El sistema debe adquirir datos de un array de sensores de temperatura a intervalos regulares con un período de adquisición seleccionable.
 - 2.2 El sistema debe adquirir datos de un anemómetro a intervalos regulares con un período de adquisición seleccionable.
 - 2.3 El sistema debe almacenar los datos de temperatura y velocidad de viento adquiridas junto con una marca de tiempo identificatoria en un medio físico no volátil.
 - 2.4 El sistema debe poder operar con dos perfiles de consumo de energía: máximo desempeño y mínimo consumo de energía, respectivamente.
 - 2.5 El sistema debe contar con una interfaz serie cableada que permita realizar operaciones de configuración y mantenimiento.
3. Requerimientos de verificación:
 - 3.1 Se debe generar una matriz de trazabilidad entre la Memoria Técnica y los requerimientos.
 - 3.2 Se debe generar una matriz de trazabilidad entre las pruebas de integración y los requerimientos.
4. Requerimientos de validación:
 - 4.1 Se debe generar una matriz de trazabilidad entre el documento de casos de prueba y los requerimientos.

2.2. Planificación

La planificación completa del proyecto puede encontrarse publicada en la web del Laboratorio de Sistemas Embebidos de FIUBA [5].

A los fines de facilitar la comprensión del trabajo realizado, se detallan en la tabla 2.1 las etapas del proyecto junto con la cantidad de horas destinadas y los hitos a alcanzar en cada una de ellas. Puede observarse que el proyecto insume 600 horas de trabajo en total.

TABLA 2.1: Etapas principales del proyecto. Se detallan de las horas planificadas y los hitos a alcanzar en cada etapa.

Etapa	Horas	Hitos
Documentación y análisis	100	Plan de trabajo Presentación de plan de trabajo
Diseño e implementación	340	Documentación de submódulos
Verificación y validación	60	Reporte de pruebas unitarias Reporte de pruebas de integración Reporte de casos de prueba
Proceso de cierre	100	Memoria Técnica Presentación de Trabajo Final

Para cada una de las etapas del proyecto, se realizó un desglose de tareas procurando que la duración de cada una no supere las 40 horas, para mejorar el proceso de control y seguimiento.

Las tareas identificadas se arreglaron esquemáticamente en el diagrama de *Activity on Node* que se ilustra en la figura 2.1. Se utiliza un mismo color para identificar las distintas tareas que componen una misma etapa del proyecto según la tabla 2.1. En color amarillo se destacan las tareas de la etapa de documentación y análisis; en celeste las tareas de diseño e implementación; en verde las tareas de verificación y validación y finalmente, en color rojo, las tareas que componen el proceso de cierre.

En el diagrama, los tiempos de duración de las tareas están representados con la variable t y expresados en horas. Asimismo, las tareas poseen un código numérico único que fue usado para realizar la trazabilidad de los requerimientos en las distintas etapas del proyecto.

Cabe destacar que dentro de las tareas de planificación, se incluyó la definición de casos de prueba. Se considera que fue una decisión acertada por el mejor entendimiento que se logró sobre las características de diseño que debía contener el sistema. Asimismo, esto permitió descartar características que se pensaban implementar pero que no respondían al cumplimiento de ningún requerimiento.

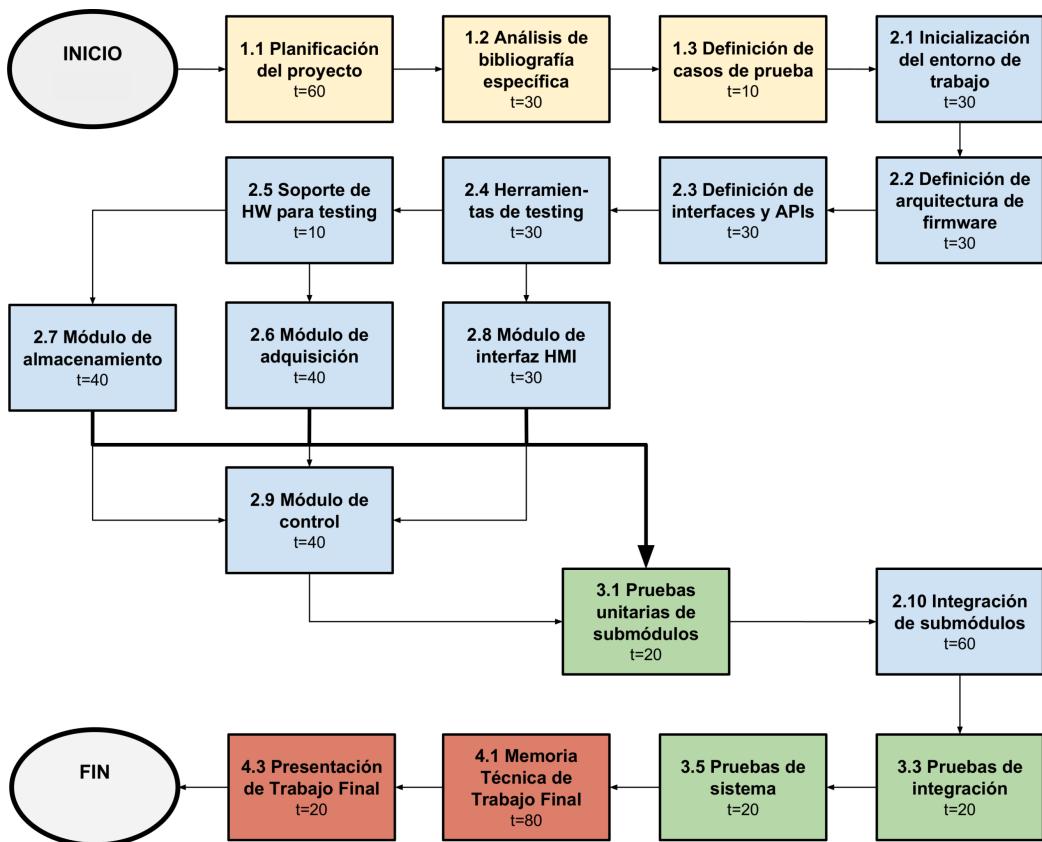


FIGURA 2.1: Diagrama *Activity on Node*. En colores puede distinguirse: en amarillo la etapa de documentación y análisis preliminar; en azul la etapa de diseño e implementación; en verde la etapa de verificación y validación y en rojo el proceso de cierre. El tiempo t está expresado en horas.

2.3. Metodología

En esta sección se describen los aspectos metodológicos relevantes que se aplicaron durante el desarrollo del trabajo.

2.3.1. Control de versiones

Se adoptó un modelo de desarrollo creado por Vincent Driessen llamado “*A successful Git branching model*” [6]. El modelo está basado en la herramienta de control de versiones *git* y consiste en un conjunto de procedimientos para ordenar y sistematizar el flujo de trabajo. Este modelo propone utilizar un repositorio considerado a los fines prácticos “central” (en *git* todos los repositorios son idénticos) llamado *origin*. Todos los desarrolladores trabajan contra este repositorio central con las operaciones típicas de *push* y *pop*.

Adicionalmente, puede haber intercambios entre los repositorios de los distintos desarrolladores que formen un mismo equipo de trabajo. Estos intercambios pueden visualizarse en la figura 2.2, donde se esquematizan, por un lado, los posibles flujos de trabajo entre el repositorio *origin* y los distintos desarrolladores, y por el otro, entre los repositorios propios de cada desarrollador.

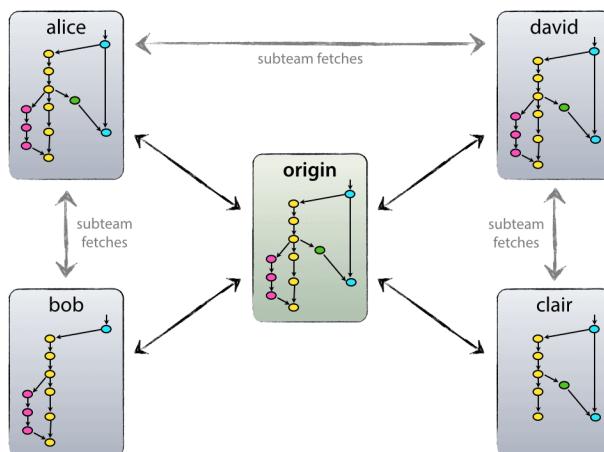


FIGURA 2.2: Esquema del flujo de trabajo entre repositorios¹.

Para la elaboración de este trabajo, donde la codificación recayó principalmente sobre una sola persona, no fueron habituales las operaciones contra un repositorio distinto de *origin*, implementado en *github*. Sin embargo, se considera la experiencia de apropiación de la metodología de trabajo muy valiosa para la formación profesional, ya que el autor de este trabajo no había tenido oportunidad de trabajar tan extensa y sistemáticamente con control de versiones previamente.

En cuanto a la estrategia de uso de ramas, siguiendo el modelo adoptado, se dispuso de dos ramas principales llamadas *master* y *develop*. En *origin/master* sólo se incluyen *commits* con versiones estables con capacidad de ser puestas en producción, es decir sobre el prototipo de manera que éste pueda operar satisfactoriamente. En *origin/develop* se encuentran los últimos cambios que integran las

¹Imagen tomada de <https://nvie.com/img/centr-decentr@2x.png>.

diferentes características ya logradas del código. Cuando la rama *develop* alcanza un punto de estabilidad y madurez suficiente se debe hacer un *merge* con *master*.

En términos generales, el esquema de ramas propuesto por el modelo de Vincent Driessen puede observar en la figura 2.3 donde se explicitan las posibles interacciones entre ramas.

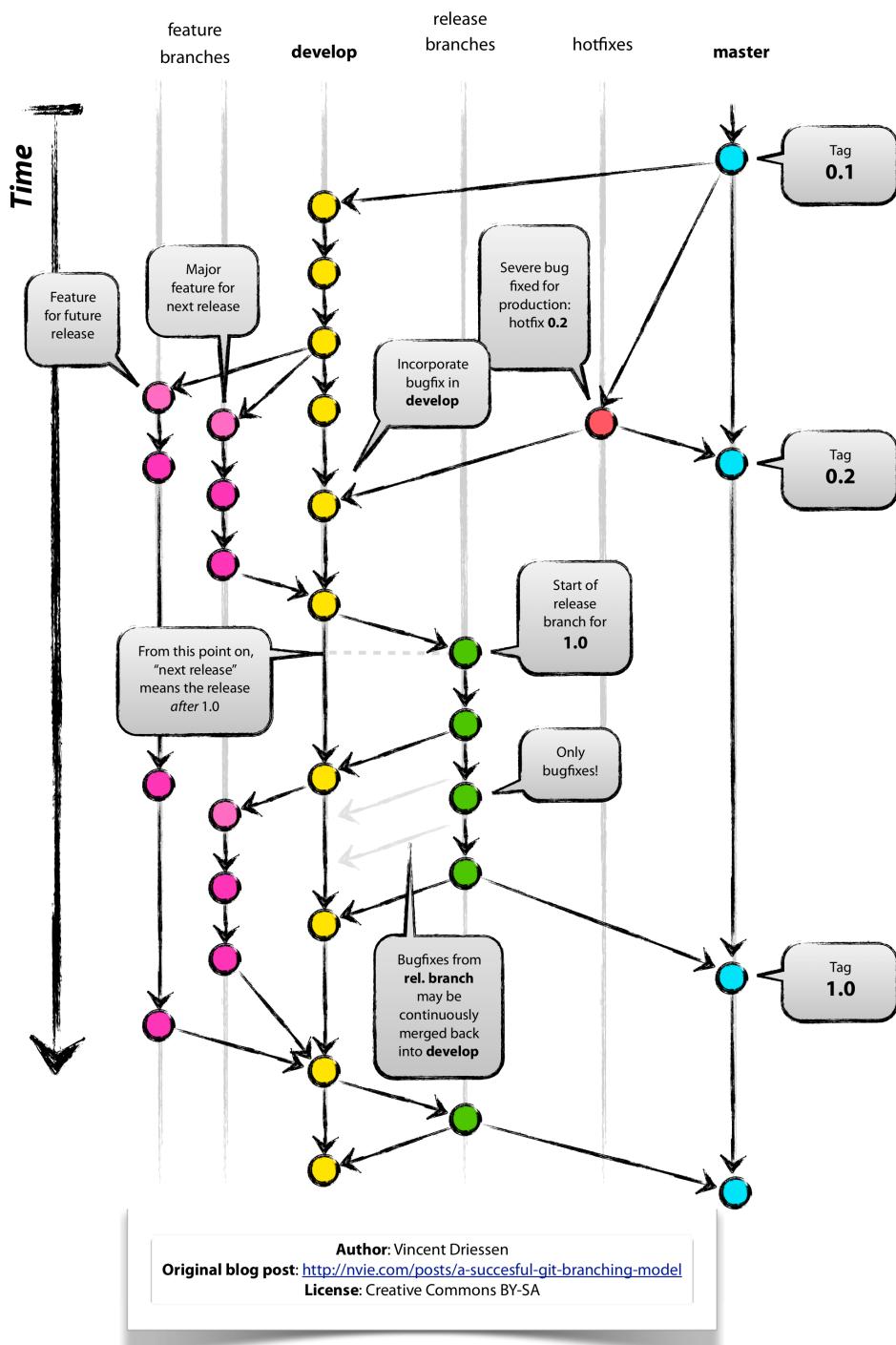


FIGURA 2.3: Modelo de ramas utilizado en git².

²Imagen tomada de <https://nvie.com/files/Git-branching-model.pdf>.

La mecánica de trabajo indica crear una nueva rama por cada característica a implementar. Cuando la característica se logra, se debe hacer *merge* con la rama *develop* cuidando que no suceda un *fastforward* y se pierdan los *commits* de la rama recién integrada.

Para este trabajo, se crearon ramas para desarrollar cada uno de los subsistemas mencionados en el diagrama en bloques de la figura 1.2 e incluidos en el alcance del trabajo, a saber:

- *sdcard* para el módulo de almacenamiento.
- *onewire* para el módulo de adquisición.
- *hmi* para el módulo de interfaz con el usuario.
- *control* para el módulo de control.
- *ceedling* para el entorno de testing.

2.3.2. Paradigma de desarrollo basado en pruebas

El paradigma de desarrollo basando en pruebas o TDD (por sus siglas en inglés, *Test Driven Development*) [7] es una práctica de programación que forma parte de las metodologías ágiles. TDD plantea invertir el orden tradicional de desarrollo en el cual primero se implementa y luego se prueba. En este sentido, este paradigma propone un proceso de desarrollo que consiste en codificar pruebas, desarrollar y refactorizar de forma iterativa el código.

Si bien no se adoptó el modelo de desarrollo TDD en forma íntegra, sí se incorporaron algunos de sus principios a la metodología de trabajo, en particular los que permiten la producción de un *firmware* modular, altamente reutilizable y preparado para el cambio, es decir escalable, conocidos como principios “SOLID”[8]:

- Principio de responsabilidad única (*Single responsibility principle*). Cada módulo debe tener una única responsabilidad.
- Principio de abierto/cerrado (*Open/closed principle*). Un módulo debe estar abierto para su extensión, pero cerrado para su modificación.
- Principio de sustitución de Liskov (*Liskov substitution principle*). Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.
- Principio de segregación de la interfaz (*Interface segregation principle*). Muchas interfaces cliente específicas son mejores que una interfaz de propósito general.
- Principio de inversión de la dependencia (*Dependency inversion principle*). Dependiendo de abstracciones, no depender de implementaciones. Esto significa que un módulo A no debe depender de otro módulo B, sino a través de una abstracción de B.

Asimismo, se tomó de TDD la filosofía de pensar primero cómo se va a probar que el código cumple los requerimientos para lograr un mejor entendimiento del sistema y aumentar la calidad del diseño.

En el capítulo 4 se recopila la documentación de casos de prueba que se usaron como entrada al diseño del *firmware* previo de su implementación.

2.3.3. Programación concurrente con Protothreads

Los Protothreads son una abstracción creada por Adam Dunkel para implementar mecanismos de programación concurrente, conocidos como multi-tarea cooperativa, en sistemas embebidos con recursos limitados [9].

Se distribuyen como una biblioteca que puede integrarse al proyecto y posibilitan trabajar con hilos de ejecución sin *stack* o co-rutinas, con mecanismos para bloquear la ejecución de una tarea sin que se produzca un cambio de contexto. Esto permite un control de flujo secuencial sin máquinas de estado complejas o soporte multi-hilo completo en arquitecturas basadas en eventos [10] [11].

Los protothreads se implementan como un conjunto de macros que el precompilador expande en tiempo de compilación.

```
struct pt { unsigned short lc; };

#define PT_THREAD(name_args)    char name_args
#define PT_BEGIN(pt)           switch(pt->lc) { case 0:
#define PT_WAIT_UNTIL(pt, c)   pt->lc = __LINE__; \
                           case __LINE__: \
                           if(!(c)) return 0
#define PT_END(pt)             } pt->lc = 0; return 2
#define PT_INIT(pt)            pt->lc = 0
```

En los algoritmos 2.1 y 2.2 se puede apreciar un ejemplo sencillo que utiliza esta herramienta y el mismo código con las macros expandidas, respectivamente. El *quid* de la implementación es la utilización de una estructura *switch* y la macro *__LINE__* para insertar el número de línea y permitir la reentrada a una parte del código. En este ejemplo, *PT_WAIT_UNTIL* se encuentra en la línea 12.

<pre>static PT_THREAD(example(struct pt *pt)) { PT_BEGIN(pt); while(1) { PT_WAIT_UNTIL(pt, counter == 1000); printf("Threshold reached\n"); counter = 0; } PT_END(pt); }</pre>	<pre>static char example(struct pt *pt) { switch(pt->lc) { case 0: while(1) { pt->lc = 12; case 12: if (!(counter == 1000)) return 0; printf("Threshold reached\n"); counter = 0; } } pt->lc = 0; return 2; }</pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

ALGORITMO 2.1:
Ejemplo de uso

ALGORITMO 2.2:
Código expandido

Se debe tener en cuenta que por la forma en que están implementados, no es posible incluir bloques de control *switch-case* dentro de los protothreads.

En el presente trabajo, se hace uso de protothreads en la codificación del protocolo de comunicación 1-wire que se describe en la subsección 3.3.1.

2.4. Arquitectura multicore

El *firmware* está desarrollado sobre la base de dos proyectos vinculados dentro IDE MCUXpresso, uno para cada *core* del microcontrolador. Para el IDE, debe haber un proyecto “maestro” que controle la ejecución del código (o al menos la secuencia de *startup*) corriendo en el otro *core*, considerado “esclavo”.

El proyecto maestro contiene un link al proyecto esclavo que produce que la imagen binaria del esclavo sea incluida en la imagen binaria del maestro, cuando el proyecto maestro es compilado [12]. De esta manera, cuando el proyecto maestro es grabado en la flash del microcontrolador, ambos proyectos son descargados a la memoria del microcontrolador.

El proyecto maestro debe ser el que se ejecuta sobre el procesador Cortex-M4 ya que el procesador Cortex-M0 permanece en estado de *reset* hasta que el otro *core* lo libera de este estado escribiendo un 0 en el bit M0SUB_RST del registro RESET_CTRL1, como se indica en el manual del microcontrolador [13].

Cuando se energiza el microcontrolador o se produce un *reset*, el *core* maestro inicia su secuencia de *startup* y es responsable de iniciar, a su vez, la secuencia de *startup* del *core* esclavo.

En las tablas 2.2 y 2.3 se muestra la asignación de bloques de memoria para los procesadores Cortex-M4 y Cortex-M0, respectivamente. Puede verse que el código de cada procesador se ubica en un bloque de memoria flash independiente, los bancos A y B de 512 kB cada uno.

Por otra parte, para evitar cualquier tipo de solapamiento en el uso de la RAM, los proyectos asociados a cada *core* se linkean de forma de utilizar exclusivamente bancos de RAM separados. En este sentido, el procesador cortex-M4 utiliza el primer bloque de RAM de 32 kB y el procesador cortex-M0 utiliza el segundo bloque de RAM de 40kB.

TABLA 2.2: Asignación de bloques de memoria para el Cortex-M4

Tipo de memoria	Nombre	Alias	Ubicación	Tamaño
Flash	MFlashA512	Flash	0x1a000000	0x80000
RAM	RamLoc32	RAM	0x10000000	0x8000

TABLA 2.3: Asignación de bloques de memoria para el Cortex-M0

Tipo de memoria	Nombre	Alias	Ubicación	Tamaño
Flash	MFlashB512	Flash2	0x1b000000	0x80000
RAM	RamLoc40	RAM2	0x10080000	0xa000

Adicionalmente, se define una zona de memoria compartida, visible por ambos procesadores para el intercambio de información. Los mecanismos de comunicación inter-procesadores (IPC, del inglés *Inter Processor Communication*) se describen en la sección 2.4.1.

```
/* Shared memory used by IPC */
#define SHARED_MEM_IPC    0x10088000
```

2.4.1. Inter Processor Communications

Para comunicar ambos procesadores se utiliza una biblioteca provista por el fabricante del microcontrolador NXP, documentada en la nota de aplicación “AN1117: Inter Processor Communications for LPC43xx” [14]. En este documento se explican mecanismos posibles para que los dos procesadores intercambien información basados en interrupciones, en colas de mensajes y en “casillas de correo”. Este último método queda excluido de esta memoria por no haber sido utilizado en el desarrollo.

Interrupciones cruzadas

El mecanismo de interrupciones cruzadas es el más simple de los tres métodos provistos. Permite que un *core* active una interrupción en el otro *core* para enviar notificaciones cuya interpretación depende y es exclusiva de la aplicación. El diseñador puede definir una función de *callback* que es ejecutada en el contexto de la rutina de servicio de la interrupción.

Para enviar señales al *core* “remoto”, el *core* “local” utiliza una instrucción dedicada SEV (*send event*) provista por la arquitectura Cortex.

Asimismo, dentro de la rutina de interrupción se habilita un *flag* para indicar que se ha recibido una notificación IPC. Esta variable *flag* puede ser usada por las aplicaciones corriendo sobre el *core* que recibe la notificación para chequear el estado de las comunicaciones.

La limpieza del *flag* se hace dentro de una sección crítica donde se deshabilitan temporalmente las interrupciones. En el Cortex-M4 se enmascaran las interrupciones con mayor prioridad y en el Cortex-M0 se deshabilitan directamente, ya que este procesador no dispone del mecanismo de enmascaramiento.

El código para generar las interrupciones cruzadas utiliza dos macros. Primero __DSB() (*Data Syncronization Barrier*) para terminar todas las transacciones de memoria pendientes y luego __SEV() (*Send Event*) para generar el envío de una señal de interrupción al otro procesador como puede verse en el algoritmo 2.3.

```
/*
 * Initiate interrupt on other processor
 * Upon calling this function generates and interrupt
 * on the other core. Ex. if called from M0 core it
 * generates interrupt on M4 core and vice versa.
 */
static void ipc_send_signal(void)
{
    __DSB();
    __SEV();
}
```

ALGORITMO 2.3: Función ipc_send_signal que permite generar una interrupción en el otro procesador.

Colas de mensajes

En el método de colas de mensajes, se deben definir dos áreas de memoria compartida, que se utilizan para almacenar los mensajes que cada procesador envía al otro. Una cola (búfer de comandos del *host*) está dedicado a los comandos enviados del procesador maestro al esclavo, y una cola (búfer de mensajes del *host*) se dedica a los mensajes que el procesador esclavo envía en respuesta al procesador maestro. La figura 2.4 muestra esquemáticamente esta configuración.

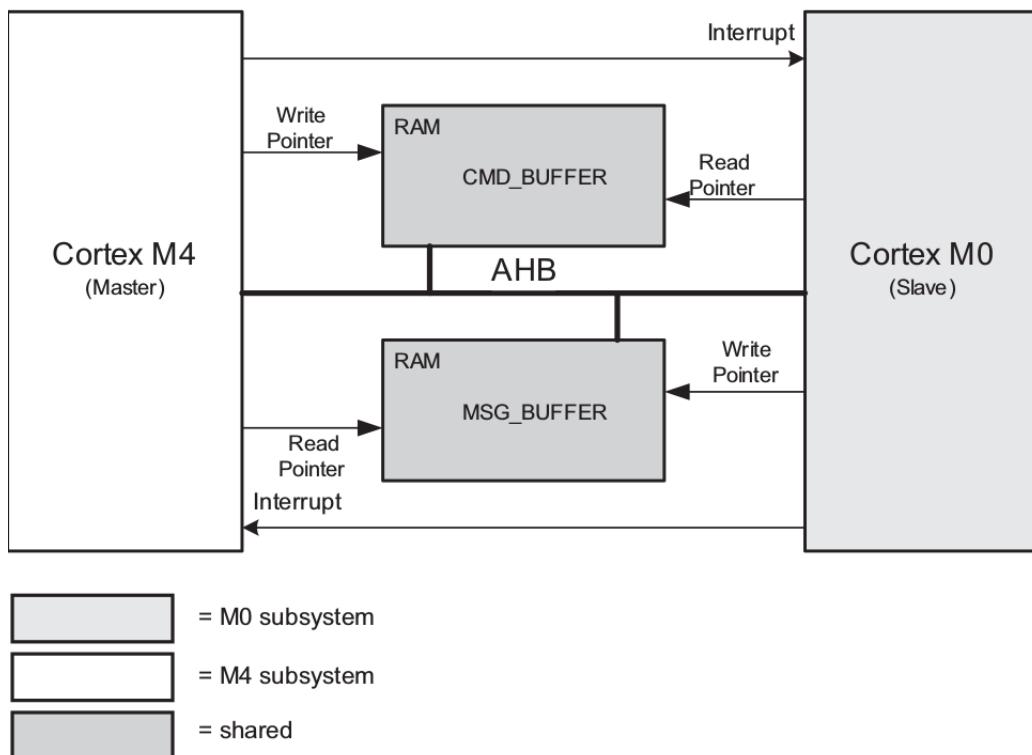


FIGURA 2.4: Esquema de comunicación entre procesadores basado en colas de mensajes³.

En el esquema propuesto, sólamente el procesador maestro puede escribir en el búfer de comandos y recibe los mensajes del esclavo leyendo el búfer de mensajes. De manera análoga, únicamente el esclavo puede escribir en el búfer de mensajes y recibe comandos leyendo el búfer de comandos.

Cuando un procesador escribe un nuevo mensaje en una cola, debe notificar al otro procesador de que hay nueva información para procesar. Para tal fin, se utiliza el mecanismo de interrupción descripto en el apartado Interrupciones cruzadas, dentro de la subsección 2.4.1.

El procesador que recibe el mensaje invoca a un despachador de eventos que busca en un vector de manejadores de eventos el que haya sido registrado para tal fin. En el algoritmo 2.4 se puede apreciar la función provista por la biblioteca IPC para despachar eventos.

³Imagen tomada del manual de usuario del microcontrolador LPC4337 [13]

```

/* This task will receive the message from the other
 * core and will invoke the appropriate callback with
 * the message
 */
static void ipcex_dispatch_task(void *unused)
{
    int ret;
    ipcex_msg_t msg;
    do {
        ret = IPC_popMsgTout(&msg, -SYS_OS_ID);
        if ((ret != QUEUE_VALID) ||
            (msg.id.pid >= IPCEX_MAX_PID)) {
            continue;
        }
        if (ipcex_callback_lookup[msg.id.pid]) {
            ipcex_callback_lookup[msg.id.pid](&msg);
        }
    } while (SYS_OS_ID);
}

```

ALGORITMO 2.4: Función despachadora de eventos de la biblioteca IPC.

SYS_OS_ID es una macro que se utiliza para identificar el tipo de Sistema Operativo incluido en la aplicación. En el caso particular de este trabajo, SYS_OS_ID es igual a 0 y esto significa que no hay Sistema Operativo. Se debe notar que SYS_OS_ID = 0 implica que el despachador de eventos se ejecuta una única vez por mensaje recibido.

La función despachadora de eventos descripta en el algoritmo 2.4 fue modificada para adaptarla a las necesidades del proyecto. En la sección 3.5 se documenta cómo se implementó esta función dentro del módulo de control de la estación de monitoreo de ruido acústico.

Para el intercambio de mensajes en el sistema, se define un nuevo tipo de dato `ipcex_msg_t`. Se trata de una estructura que contiene información que identifica al CPU y al proceso destinatarios del mensaje, definidos dentro de otra estructura anidada, junto con dos campos para datos como se observa en el algoritmo 2.5.

```

typedef struct __ipcex_msg {
    struct {
        uint16_t cpu;
        uint16_t pid;
    } id;

    uint32_t data0;
    uint32_t data1;
} ipcex_msg_t;

```

ALGORITMO 2.5: Definición de un nuevo tipo de dato `ipcex_msg_t` para intercambio de mensajes.

El tipo de datos `ipcex_msg_t`, tal como viene definido en la biblioteca IPC fue modificado como se documenta en la sección 3.5.

Capítulo 3

Diseño e Implementación

En este capítulo se presenta la arquitectura del *firmware* y el patrón de diseño usado para los módulos del sistema. Asimismo, se detallan aspectos funciones de cada módulo y se fundamentan las elecciones de los distintos componentes de hardware utilizados.

3.1. Diseño de módulos y definición de interfaces

Se optó por un diseño fuertemente modularizado en archivos, junto a un modelo de capas jerárquicas para organizar el código en distintos niveles de abstracción.

En la figura 3.1 se pueden observar, agrupadas por color, las capas utilizadas en este trabajo. En color amarillo, el paquete de drivers provistos por el fabricante del silicio; en color naranja la capa de bibliotecas; la capa de abstracción de *hardware* (HAL, por sus siglas en inglés) en color verde; y finalmente, en color celeste, la capa de aplicación con los cuatro módulos implementados.

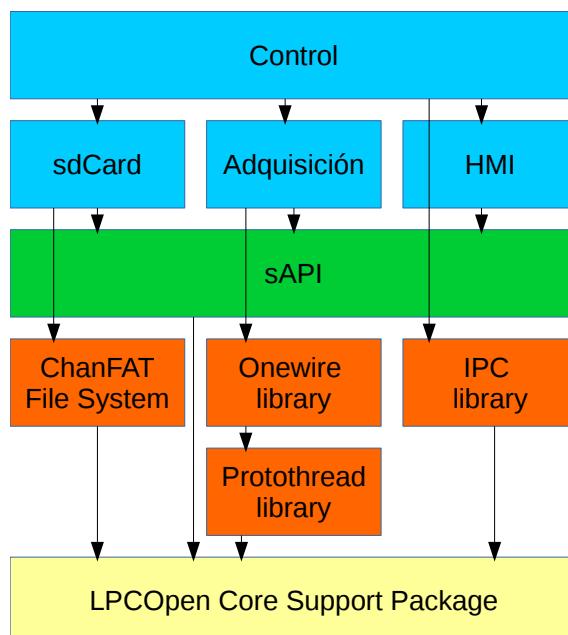


FIGURA 3.1: Estructura de capas para el *firmware*. En orden creciente de nivel de abstracción: *core support package* (amarillo), bibliotecas (naranja), *sAPI* (verde) y la capa de aplicación (celeste) con los 4 módulos implementados.

Para los módulos en la capa de aplicación, se confeccionó una matriz de trazabilidad de los requerimientos referidos al firmware, detallados en la sección 2.1.

La tabla 3.1 permite saber en qué módulo del firmware serán implementados los requerimientos funcionales del proyecto. Asimismo, permite controlar que todo este subconjunto de requerimientos sea implementado y que no haya superposición entre la funcionalidad de cada módulo.

TABLA 3.1: Matriz de trazabilidad de requerimientos funcionales con los módulos de *firmware*

Requerimiento	Control	sdCard	Adquisición	HMI
2.1 Adquirir temperatura			X	
2.2 Adquirir velocidad de viento			X	
2.3 Almacenar datos		X		
2.4 Perfiles de consumo	X			
2.5 Interfaz				X

Se definió un nuevo tipo de dato, `module_t` que contiene campos de control para los módulos. Se observa su la definición el el algoritmo 3.1.

El primer campo de `module_t` indica en qué procesador debe ejecutarse el módulo. Sigue un campo para identificar al módulo y un puntero a la función manejadora de eventos que será invocada cada vez que haya un mensaje para este módulo. Para temporizar en forma periódica la ejecución del módulo se utiliza el campo `period` y, finalmente, se incluye un campo para el estado del módulo que puede tomar uno de los siguientes valores: DISABLE, READY o PROCESSING.

```
typedef struct {
    CPUID_T coreID;
    moduleID_t moduleID;
    funcPtr_t eventHandler;
    tick_t period;
    moduleStatus_t status;
} module_t;
```

ALGORITMO 3.1: Definición de un nuevo tipo de dato `module_t`.

Toda la interacción con los módulos se realiza a través de los respectivos manejadores de eventos y una cola de mensajes como la descripta en la subsección 2.4.1. Para el intercambio de mensajes entre módulos que se encuentren en ejecución en un mismo procesador, se levantaron las restricciones de escritura sobre la propia cola de mensajes que se originalmente pesaban sobre cada procesador.

A continuación se listan los prototipos de los manejadores de eventos definidos:

- `void onewire_handler(const ipcex_msg_t * msg);`
- `void hmi_handler(const ipcex_msg_t * msg);`
- `void sdcard_handler(const ipcex_msg_t * msg);`
- `void control_handler(const ipcex_msg_t * msg);`

Puede notarse que todos los manejadores reciben un mismo tipo de parámetro, un puntero constante a un mensaje de tipo `ipcex_msg_t`. Cada módulo es responsable de interpretar los dos campos de datos que contiene el mensaje. Normalmente el primer campo, `signal` define un comando o señal y el segundo, `param` se utiliza opcionalmente para el envío de parámetros u opciones.

Para todos los módulos se utilizó un modelo de *firmware* basado en máquinas de estados finitos (MEF) jerárquicas, donde una MEF principal controla la lógica de funcionamiento con el mayor nivel de abstracción. Los diferentes estados, a su vez, pueden o no estar modelados con MEFs dependiendo de la conveniencia de esto último.

Se definió una arquitectura de máquina de estados finitos principal lo más genérica posible de forma que pueda ser compartida por los distintos módulos. En este sentido, todos los módulos poseen un estado `DISABLE`, `INIT`, `IDLE`, `CONFIG` y `CHECK`. La funcionalidad que implemente cada uno de estos estados será propia del módulo donde estén definidos.

Todos los módulos inician en estado `DISABLE`. Adicionalmente, la máquina de estados finitos podrá tener otros estados que sean específicos para lograr sus propósitos funcionales.

En la figura 3.2 se muestra un diagrama de la máquina de estados finitos genérica, diseñada para ser usada como base en cada uno de los módulos que componen el sistema.

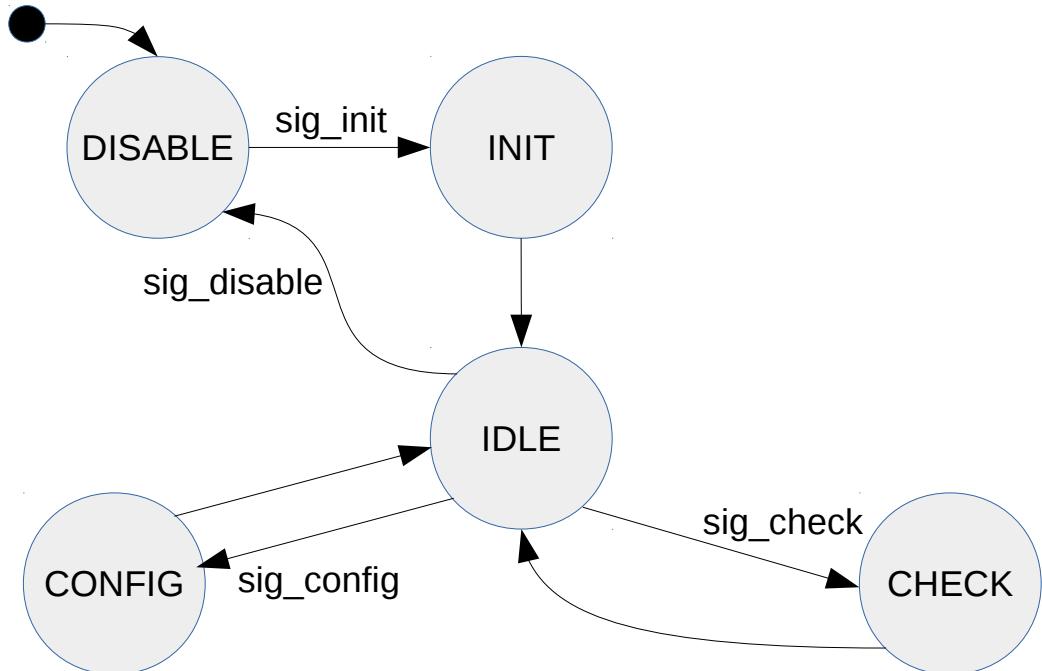


FIGURA 3.2: Diseño de máquina de estados finitos genérica para el control de la lógica de los módulos

3.2. Módulo de almacenamiento

El propósito del módulo de almacenamiento es proveer al sistema de una interfaz para operar con un medio de almacenamiento no volátil que permita guardar en forma permanente los datos registrados por el módulo de adquisición y eventualmente un log con información de *debug*.

3.2.1. Medio físico

Se evaluaron distintas opciones de medios físicos. En la tabla 3.2 se recopilan las alternativas analizadas y se especifica un orden de magnitud para la capacidad de almacenamiento posible, el tipo de interfaz con el microcontrolador y el protocolo que debe implementarse en el *firmware* para su utilización.

TABLA 3.2: Alternativas de medios físicos evaluados.

Medio físico	Capacidad	Interfaz	Protocolo
USB Mass Storage Device	~10 Gb	USB 2.0	USB
Tarjeta de memoria microSD	~10 Gb	Micro SD	SSP
Disco de estado sólido (SSD)	~100 Gb	SATA III	SATA
Disco duro mecánico (HHD)	~1000 Gb	SATA III	SATA

Teniendo en cuenta criterios de costos y simplicidad de interacción con la CIAA-NXP, se decidió utilizar un lector de tarjetas microSD con comunicación sobre un puerto *Synchronous Serial Port* (SSP). Este protocolo es compatible con el protocolo SPI y utiliza un bus de 4 cables con las señales SCK, SSEL, MISO y MOSI. Asimismo, el soporte seleccionado minimiza el consumo de energía comparado con las otras alternativas, lo cual lo hace deseable para la aplicación.

El lector de tarjetas utilizado se presenta en la figura 3.3, donde puede verse esquemáticamente el diagrama de conexionado eléctrico entre la CIAA-NXP y el lector (figura 3.3a) y una vista superior del módulo de hardware (figura 3.3b). Cabe notar que al haber un único dispositivo “escalvo” en el bus SSP, la señal de *chip select* (CS) se conecta con GND, lo que implica que el dispositivo está permanente seleccionado.

CIAA-NXP

+3.3V	->	+3.3V
GND	->	CS
SPI_MOSI	->	MOSI
SPI_SCK	->	SCK
SPI_MISO	->	MISO
GND	->	GND

(A)

Lector de tarjetas



(B)

FIGURA 3.3: (A) Diagrama de conexionado eléctrico y (B) Lector de tarjetas SD utilizado.

Para la utilización del lector de tarjetas se hizo uso de la biblioteca FatFs, desarrollada por chaN [15]. FatFs es un módulo de sistema de archivos FAT/exFAT genérico para sistemas embebidos limitados en recursos. En la figura 3.4 se esquematizan las interfaces de la biblioteca en una aplicación típica.

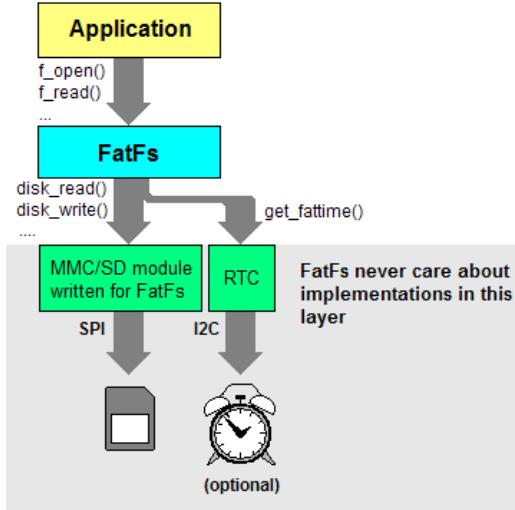


FIGURA 3.4: Diagrama de capas con la ubicación de la biblioteca fatFs en el sistema y sus interfaces¹.

El código está escrito en ANSI C y es *software libre* bajo licencia estilo BSD [16].

3.2.2. Máquina de Estados Finitos

Se presenta el diagrama de estado de la MEF principal del módulo en la figura 3.5, donde puede apreciarse el punto de entrada con un círculo negro, los estados que puede tomar la máquina y las señales que provocan los cambios de estado. Se omiten del gráfico las salidas del sistema por simplicidad.

Debe notarse que al energizarse el sistema o luego de un *reset*, el módulo se encuentra deshabilitado, con la MEF en el estado `DISABLE`, del cual sólo puede salir si se recibe una señal de inicialización.

Una vez inicializado, el módulo se encontrará la mayor parte del tiempo en el estado `IDLE` a la espera de un comando válido.

Cuando el módulo esté realizando alguna operación de lectura, escritura o actualización sobre la tarjeta SD, el estado del módulo tendrá el valor `PROCESSING` para indicarle al módulo de control que debe tener acceso a tiempo de CPU para poder completar las operaciones pendientes.

Todas las operaciones del módulo desde que es inicializado terminan incondicionalmente en el estado `IDLE`, en donde el valor de la variable que registra el estado del módulo cambia de `PROCESSING` a `READY`.

En la tabla 3.3 se describen los posibles estados de la MEF junto con las señales para alcanzarlos. Asimismo se explicitan las acciones y actividades más destacables que se realizan en cada uno de ellos.

¹Imagen tomada de <http://elm-chan.org/fsw/ff/doc/appnote.html>.

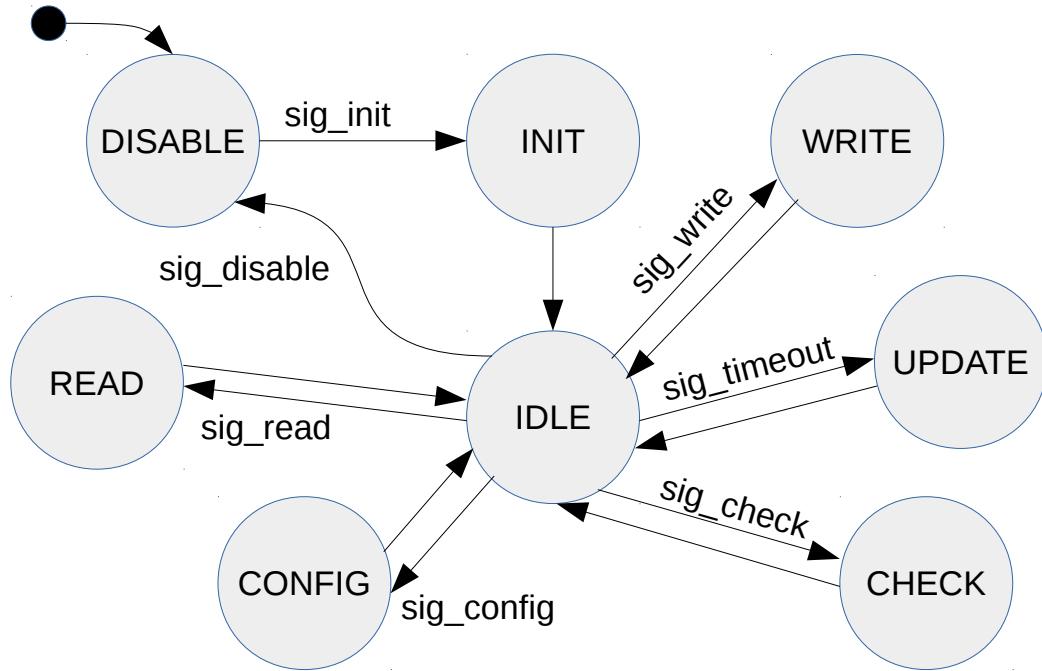


FIGURA 3.5: Máquina de estados finitos principal del módulo de almacenamiento.

TABLA 3.3: Descripción de los estados de la MEF principal del módulo de almacenamiento.

Estado	Señal para cambio	Acciones y actividades
DISABLE	Reset o sig_disable	Desinicializar el controlador SPI. Desinicializar la biblioteca fatFS. Desinicializar el módulo de almacenamiento. Cambiar el estado del módulo de READY a DISABLE.
INIT	sig_init	Inicializar el controlador SPI. Inicializar la biblioteca fatFS. Iniciar el módulo de almacenamiento. Cambiar el estado del módulo de DISABLE a READY.
IDLE	transición incondicional	Cambiar estado del módulo de PROCESSING a READY.
CONFIG	sig_config	Realizar cambios de configuración al módulo.
CHECK	sig_check	Realizar autochequeo e informar al módulo de control
UPDATE	sig_timeout	Ejecutar tarea periódica de actualización de la tarjeta SD.
WRITE	sig_write	Obtener timestamp y escribir datos en la tarjeta SD.
READ	sig_read	Leer datos en la tarjeta SD

3.3. Módulo de adquisición

El propósito del módulo de adquisición es proveer al sistema de una interfaz para operar los distintos sensores que puedan conectarse a la estación. Originalmente, se contempló el uso de uno o varios termómetros y un anemómetro para medir temperatura del agua y velocidad del viento en superficie, respectivamente. Adicionalmente, se contempla incorporar hidrofónicos para registrar el nivel de ruido submarino, principal objetivo de la estación de medición.

Se mencionó en la sección 1.3, que los hidrofónicos quedaron fuera del alcance del proyecto en esta etapa del desarrollo. De igual manera, el control del anemómetro quedará fuera de la implementación ya que se optó por priorizar el requerimiento implícito de cumplir con la fecha de entrega. Se retoma este punto en el capítulo 5, en la sección 5.1.2, donde se discuten sus implicancias.

Sin pérdida de generalidad, la arquitectura de módulo adoptada puede utilizarse para incorporar los sensores no implementados, en una segunda iteración.

3.3.1. Sensor de temperatura

Se evaluaron distintas opciones para el sensor de temperatura que estuvieran disponibles en el mercado local. Se recopila en la tabla 3.4 la información relevada para los sensores candidatos, donde puede verse fabricante y modelo de cada dispositivo junto con las principales características de interés para la aplicación.

Cabe destacar que la limitante para el rango de temperatura no es el propio sensor sino el plástico que recubre los hilos conductores que lo conectan al microcontrolador, motivo por el cual todas las alternativas evaluadas poseen rangos equivalente. Se utiliza en la tabla un código sencillo para indicar un orden de magnitud del costo relativo del sensor con \$ para costo bajo, \$\$ para costo medio y \$\$\$ para costo alto.

TABLA 3.4: Alternativas de sensor de temperatura evaluadas.

Fabricante	Sensor	Tipo	Precisión (°C)	Rango (°C)	Interfaz	Costo
Maxim Integrated	DS18B20	Termómetro digital	± 0.5	-55 a +125	1-wire	\$
Genérico	IM120628010	Termistor NTC	1%	-25 a +125	1-wire	\$
Texas Instruments	LM35	Integrated Circuit	± 0.5	-55 a +150	Analog	\$
Altas Scientific	PT-1000A	RTD (resistance temperature detector)	± (0.15 + 0.002*t)	-55 a +125	Analog	\$\$\$

El sensor elegido es el termómetro digital DS18B20 del fabricante Maxim Integrated [17]. Los motivos de la elección fueron principalmente la facilidad de uso en un entorno embebido de recursos limitados, la disponibilidad de documentación completa y detallada junto con numerosas notas de aplicación y el bajo costo del dispositivo.

En la figura 3.6 se puede apreciar un diagrama en bloques del sensor, en donde destaca la interfaz de 3 cables con una línea bidireccional de datos (DQ) y dos cables para la alimentación (GND y V_{DD}).

El sensor admite una configuración de alimentación “parásita”, en la cual se energiza desde la línea de datos DQ, lo que habilita a prescindir de un cable a costa de mayores requerimientos para la temporización de las comunicaciones y la utilización de un transistor a modo de “*strong pull-up*” (conectado el bus de datos directamente a V_{DD}).

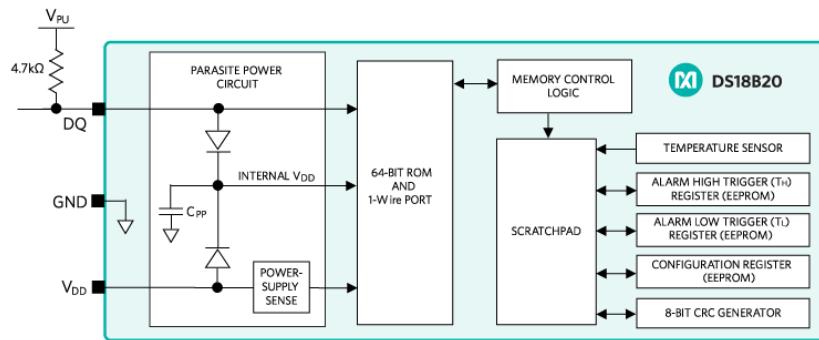


FIGURA 3.6: Diagrama en bloques del sensor de temperatura².

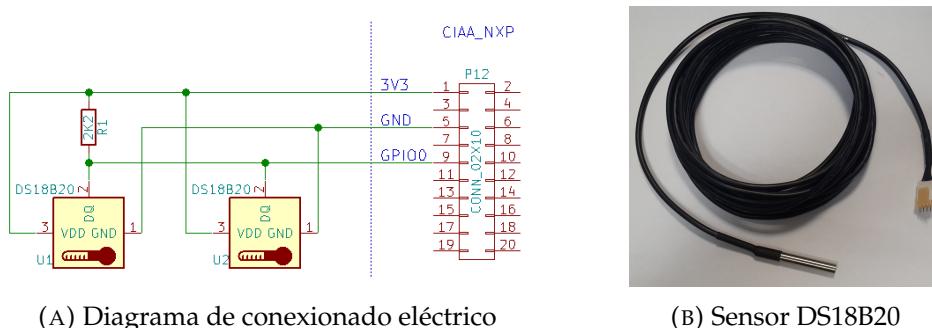
Cada sensor posee un código de identificación único de 64 bits que permite operar con múltiples sensores sobre un mismo bus de tipo 1-wire. Mediante un algoritmo de descubrimiento se pueden identificar los sensores conectados a un mismo bus y controlarlos independientemente.

La capacidad para identificar cuántos sensores hay conectados en cada momento y poder interrogarlos individualmente se utilizó para incorporar una funcionalidad de autochequeo, en la cual el microcontrolador es capaz de detectar si un sensor ha dejado de funcionar correctamente.

Otras características destacables son la posibilidad de configurar la resolución entre 9 y 12 bits, el uso de CRC en las comunicaciones y la capacidad para definir umbrales de alarma para los valores de temperatura medidos.

Este sensor se comercializa con distintas longitudes de cable, dentro de un encapsulado metálico sumergible. Para este trabajo se adquirieron dos sensores sumergibles con 5 metros de cable y 2 metros de cable, respectivamente.

Los dos sensores operan sobre una misma línea de datos con una resistencia de *pull-up* de 2,2 KΩ, como puede verse en el diagrama de conexionado eléctrico de la figura 3.7a. El encapsulado sumergible puede apreciarse en la figura 3.7b.



²Imagen tomada de <https://www.maximintegrated.com/en/images/qv/2812.png>

Para el control de los dispositivos se adaptó una implementación de controlador de bus 1-wire para microcontroladores de la familia LPC1343, desarrollada en forma de biblioteca por James Harwood [18]. En esta biblioteca se hace uso de las técnicas de programación concurrente descritas en el capítulo 2, subsección 2.3.3.

3.3.2. Máquina de Estados Finitos

En forma análoga a lo presentado en la subsección 3.2.2, se muestra aquí el diagrama de estado de la MEF principal del módulo en la figura 3.8, donde puede apreciarse el punto de entrada con un círculo negro, los estados que puede tomar la máquina y las señales que provocan los cambios de estado. Se omiten del gráfico las salidas del sistema por simplicidad.

Igualmente, debe notarse que cuando el sistema es energizado o luego de un *reset*, el módulo se encuentra deshabilitado, con la MEF en el estado DISABLE, del cual sólo puede salir si se recibe una señal de inicialización.

Una vez inicializado, el módulo se encontrará la mayor parte del tiempo en el estado IDLE a la espera de un comando válido.

Cuando el módulo esté realizando alguna operación de adquisición, configuración o autochequeo sobre los sensores conectados, el estado del módulo tendrá el valor PROCESSING para indicarle al módulo de control que debe tener acceso a tiempo de CPU para poder completar las operaciones pendientes.

Todas las operación del módulo desde que es inicializado terminan incondicionalmente en el estado IDLE, en donde el valor de la variable que registra el estado cambia de PROCESSING a READY.

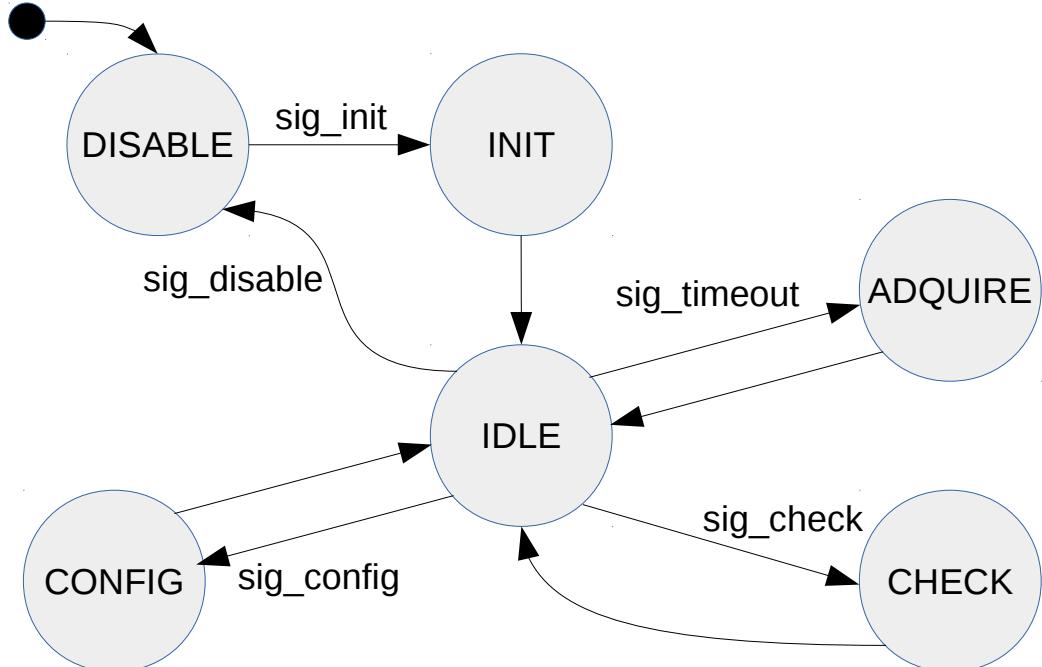


FIGURA 3.8: Máquina de estados finitos principal del módulo de adquisición de temperatura

Las principales acciones y actividades de cada estado se describen en la tabla 3.5. Asimismo, se indican las señales válidas que puede recibir la MEF junto con las transiciones que éstas provocan.

TABLA 3.5: Descripción de los estados de la MEF principal del módulo de adquisición. Se indican las señales que puede recibir junto con las acciones más relevantes de cada estado.

Estado	Señal para cambio	Acciones y actividades
DISABLE	Reset o sig_disable	Desinicializar el TIMER0. Deshabilitar la interrupción TIMER0_IRQn. Desinicializar el GPIO 3[0]. Cambiar el estado del módulo de READY a DISABLE.
INIT	sig_init	Inicializar el TIMER0. Habilitar la interrupción TIMER0_IRQn. Inicializar el GPIO3[0]. Cambiar el estado del módulo de DISABLE a READY.
IDLE	transición incondicional	Cambiar estado del módulo de PROCESSING a READY.
CONFIG	sig_config	Realizar cambios de configuración al módulo.
CHECK	sig_check	Realizar autochequeo e informar al módulo de control
ADQUIRE	sig_timeout	Ejecutar tarea periódica de adquisición de datos

3.4. Módulo interfaz máquina-hombre

El propósito del módulo interfaz máquina-hombre (HMI, por sus siglas en inglés) es proveer al sistema de un medio de interacción con un operador humano. Las posibles interacciones se realizan a través de una interfaz de comandos interactiva y contempla operaciones de configuración y cambio de parámetros sobre los módulos y operaciones de mantenimiento y depuración del sistema.

La interfaz se implementa desde el microcontrolador sobre un puerto serie de tipo *Universal Asynchronous Receiver-Transmitter* o UART a través de un conversor RS232/USB de la firma FTDI incorporado en la CIAA-NXP, hasta llegar finalmente a la PC del operador humano.

El código del módulo incluye directivas de compilación condicional que permiten cambiar la cantidad y el nivel de detalle de los mensajes que el sistema envía a través de la interfaz. Esta opción debe definirse en tiempo de compilación y no puede ser modificada en tiempo de ejecución, es decir, una vez que el sistema esté en funcionamiento.

Mediante una directiva `DEFINE` para el precompilador se puede definir el símbolo `DEBUG_ENABLE` en el archivo `board_api.h`. Si esta macro se encuentra definida, se habilitan funciones de depuración que envían mensajes a través de la UART como puede verse en el algoritmo 3.2.

Adicionalmente, si se define la MACRO `DEBUG_SEMIHOSTING` se pueden utilizar los recursos de entrada/salida de la PC host que esté depurando el sistema embebido.

Si la MACRO DEBUG_ENABLE no es definida, las funciones no se incluyen en el código, ahorrando recursos al sistema.

```
#if defined(DEBUG_ENABLE)

# if defined(DEBUG_SEMIHOSTING)

#define DEBUGINIT()
#define DEBUGOUT(...) printf(__VA_ARGS__)
#define DEBUGSTR(str) printf(str)
#define DEBUGIN() (int) EOF

#else

#define DEBUGINIT() Board_Debug_Init()
#define DEBUGOUT(...) printf(__VA_ARGS__)
#define DEBUGSTR(str) Board_UARTPutSTR(str)
#define DEBUGIN() Board_UARTGetChar()

#endif /* defined(DEBUG_SEMIHOSTING) */

#else

#define DEBUGINIT()
#define DEBUGOUT(...)
#define DEBUGSTR(str)
#define DEBUGIN() (int) EOF

#endif /* defined(DEBUG_ENABLE) */

```

ALGORITMO 3.2: Macros para habilitar/deshabilitar mensajes de depuración del código.

Para acceder a la interfaz se debe utilizar una terminal serie tipo screen [19] en configuración 8N1 a 115200 baudios.

Existen emuladores de terminales serie como *cute.com*, de gran difusión, que presentan el inconveniente de no implementar funciones de control definidas en los estándares ANSI X3.64 (ISO 6429) e ISO 2022 por lo que la visualización de la interfaz en estas consolas puede ser defectuosa y no se recomienda su uso.

3.4.1. Interfaz

El propósito de la interfaz desarrollada es servir como prueba de concepto e incluye un mínimo de funcionalidad para cumplir los requerimientos relacionados a la controlabilidad de la estación, principalmente en lo referido al período de muestreo de los sensores del módulo de adquisición.

Cabe destacar que el valor de esta interfaz no está en su estética ni en su facilidad de uso, sino en la arquitectura subyacente que permite obtener información sistema y operar sobre sus distintos componentes.

Una primera versión de la interfaz operativa se presenta en la figura 3.9, donde pueden apreciarse cuatro zonas funcionales identificadas con recuadros de color rojo, a saber: encabezado, menú de opciones, línea de comandos y barra de estado.



FIGURA 3.9: Vista de la pantalla principal de la interfaz. Se destacan con recuadros rojos cuatro secciones dentro de la pantalla: encabezado, menú de opciones, línea de comandos y barra de estado.

En el encabezado puede observarse el nombre de la estación EAMMRA, por las siglas de “Estación Autónoma Marítima de Medición de Ruido Ambiente” junto en una identificación del número de revisión de la interfaz, en el caso de la figura 3.9, r1.0.

En la zona asociada al menú de opciones se presentan los comandos que puede introducir el operador. En cada pantalla, las opciones que se muestran cambian según el contexto.

Se realiza un eco de los comandos ingresados, que aparecen en la línea de comandos para confirmación del usuario. Asimismo, si se ingresa un carácter que no sea parte de las opciones disponibles, el sistema lo informa a continuación del eco con el mensaje “no es un comando valido.”

Los diferentes módulos pueden enviar mensajes para el usuario que aparecerán en la sección indicada como barra de estado. De igual manera, cuando el sistema requiera confirmación por parte del usuario, se exhibirá un mensaje para tal fin en esta sección.

Desde el menú principal se puede acceder al modo configuración para interactuar directamente con los módulos del sistema. Se presenta la pantalla principal de este modo en la figura 3.10. Para acceder a las opciones de un módulo en particular se debe ingresar el número asociado a éste en el menú de opciones. A excepción de la pantalla principal, todas las pantallas cuentan con una opción (X) para volver al menú anterior.

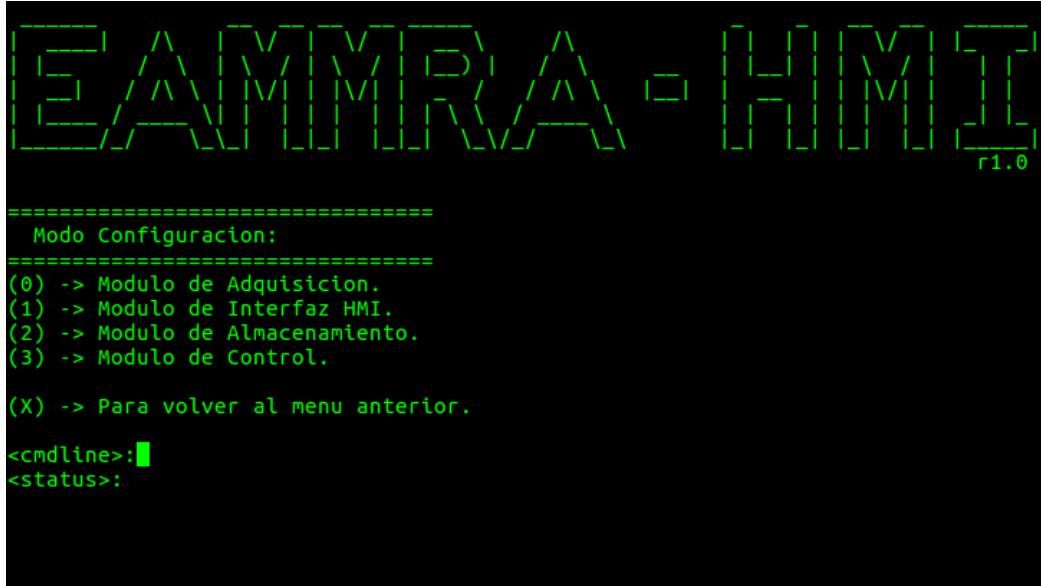


FIGURA 3.10: Vista de la pantalla principal del modo configuración de la interfaz máquina-hombre.

Sin pérdida de generalidad, ya que todos los módulos comparten las mismas opciones, se presenta en la figura 3.11 una vista de detalle de la pantalla de configuración del módulo de adquisición, también llamado módulo 1-WIRE por el tipo de sensor que controla. Se puede observar la opción de configuración elegida en la línea de comandos y el pedido de confirmación en la barra de estado.

Se contempla diferenciar a futuro las opciones de cada módulo acorde se agregue funcionalidad a los mismos. Las opciones actualmente disponibles son las siguientes:

- (D) Habilitar / Deshabilitar: permite intercambiar el estado del módulo entre READY y DISABLED. Para tal fin, si el módulo se encuentra en estado READY, se le envía una señal sig_disable. En caso que se encuentre en estado DISABLED se le envía una señal sig_init. Antes de enviar la señal correspondiente se informa al usuario el estado actual del módulo y se pide que confirme o cancele la operación mediante el ingreso de 's' o 'n', respectivamente.
- (I) Info del modulo: muestra el estado actual del módulo en la barra de estado. Las respuestas posibles son READY, PROCESSING y DISABLED
- (C) Configurar: permite cambiar la configuración del módulo. En esta etapa del desarrollo se implementó únicamente la posibilidad de cambiar el tiempo con que se ejecuta la tarea periódica, si la hubiera.
- (A) Autocomprobación: permite realizar una comprobación interna de funcionamiento mediante el envío de una señal sig_check al módulo. En esta etapa del desarrollo la funcionalidad está implementada con una función mock que devuelve un mensaje en la barra de estado.
- (X) para volver al menú anterior: cambia el menú a la pantalla principal del modo configuración.



```

=====
  Modulo 1-WIRE:
=====
(D) -> Habilitar / Deshabilitar.
(I) -> Info del modulo.
(C) -> Configurar.
(A) -> Autocomprobacion.

(X) -> Para volver al menu anterior.

<cmdline>:D
<status>:[1-WIRE] status: READY
Esta seguro que desea cambiar el estado del modulo (s/n)█

```

FIGURA 3.11: Pantalla de configuración del módulo de adquisición con una operación de deshabilitación en curso.

Asimismo, desde el menú principal se puede acceder a un modo *debug* con opciones de depuración y mantenimiento como puede observarse en la figura 3.12.



```

=====
  Modo Debug:
=====
(E) -> Estado de modulos.
(A) -> Comprobacion de modulos
(L) -> Ver log del sistema.

(X) -> Para volver al menu anterior.

<cmdline>:█
<status>:

```

FIGURA 3.12: Vista de la pantalla principal del modo debug de la interfaz máquina-hombre.

Las opciones para este modo son:

- (E) Estado de modulos: muestra el estado actual de todos los módulo del sistema en la barra de estado.
- (A) Comprobación: permite realizar una comprobación interna de funcionamiento a todos los módulos. En esta etapa del desarrollo la funcionalidad está implementada con una función *mock* que devuelve un mensaje en la barra de estado y registra los resultados en el log del sistema.

- (L) Ver log del sistema: permite realizar un *dump* del log del sistema en la consola. Esto significa que se imprime en la terminal serie todo el contenido del archivo de log ubicado en la tarjeta de memoria microSD. Presionando cualquier tecla se limpia la pantalla y se vuelve al menú principal del modo *debug*.
- (X) para volver al menú anterior: cambia el menú a la pantalla principal del modo *debug*.

En la tabla 3.6, se recopilan los comandos que se pueden ingresar por la interfaz en el modo configuración junto con una descripción de la funcionalidad que activa en el sistema. Asimismo, se explicita si la función se encuentran implementada o maquetada.

TABLA 3.6: Opciones disponibles en el modo configuración de la interfaz. Se indica el comando para seleccionar la opción y si ésta se encuentra implementada o maquetada.

Opción	Descripción	Implementada	Maquetada
D	Habilitar / Deshabilitar el módulo. Se informa el estado actual Se pide confirmación del usuario	X	
I	Informa el estado del módulo.	X	
C	Configurar el módulo Permite cambiar el valor de period dentro de la estructura de control del módulo	X	
A	Autocomprobación del módulo. Informa en la barra de estado el resultado de la verificación.		X

En forma análoga, se presenta en la tabla 3.7, una recopilación de los comandos que se pueden ingresar en el modo debug junto con una descripción de la misma. Asimismo, se explicita si la función se encuentran implementada o maquetada.

En los módulos que tiene funciones maquetadas se reemplazó la llamada a la función que debería realizar la implementación por el envío de un mensaje generado en el propio módulo hacia la interfaz HMI. De esta manera se puede comprobar el correcto funcionamiento de la interfaz y el circuito de comunicación entre módulos, es decir, que:

- el comando es correctamente recibido por la interfaz,
- la interfaz envía correctamente una señal al módulo correspondiente,
- el módulo correspondiente recibe la señal y contesta con un mensaje que se visualiza en la barra de estado de la interfaz.

TABLA 3.7: Opciones disponibles en el modo debug de la interfaz. Se indica el comando para seleccionar la opción y si ésta se encuentra implementada o maquetada.

Opción	Descripción	Implementada	Maquetada
E	Informa el estado actual de todos los módulos en la barra de estado	X	
A	Autocomprobación de todos los módulos. Se registra el resultado en el log del sistema		X
L	Listar el contenido del log del sistema. Se limpia presionando cualquier tecla	X	

Se contempla incluir en versiones futuras, además de cambios estilísticos, la implementación completa de funcionalidades que en la presente versión se encuentran maquetadas. Si bien estás y otras características son deseable para la usabilidad de la interfaz y la operación de la estación, no formaban parte de los requerimientos enumerados en la sección 2.1.

3.4.2. Máquina de estados finitos

El diagrama de estado de la MEF principal del módulo puede apreciarse en la figura 3.13, donde se observa el punto de entrada con un circulo negro, los estados que puede tomar la máquina y las señales que provocan los cambios de estado. Se omiten del gráfico las salidas del sistema por simplicidad.

Igualmente, debe notarse que cuando el sistema es energizado o luego de un *reset*, el módulo se encuentra deshabilitado, con la MEF en el estado **DISABLE**, del cual sólo puede salir si se recibe una señal de inicialización.

Una vez inicializado, el módulo se encontrará la mayor parte del tiempo en el estado **IDLE** a la espera de un comando válido para procesar.

Cuando el módulo esté realizando alguna operación de procesamiento de comandos recibidos por el usuario, configuración o autochequeo, el estado en la del módulo tendrá el valor **PROCESSING** para indicarle al módulo de control que debe tener acceso a tiempo de CPU para poder completar las operaciones pendientes.

Todas las operación del módulo desde que es inicializado terminan incondicionalmente en el estado **IDLE**, en donde el valor de la variable que registra el estado cambia de **PROCESSING** a **READY**.

Se describen las principales acciones y actividades de cada estado en la tabla 3.8. Asimismo, se indican las señales válidas que puede recibir la MEF junto con las transiciones que éstas provocan.

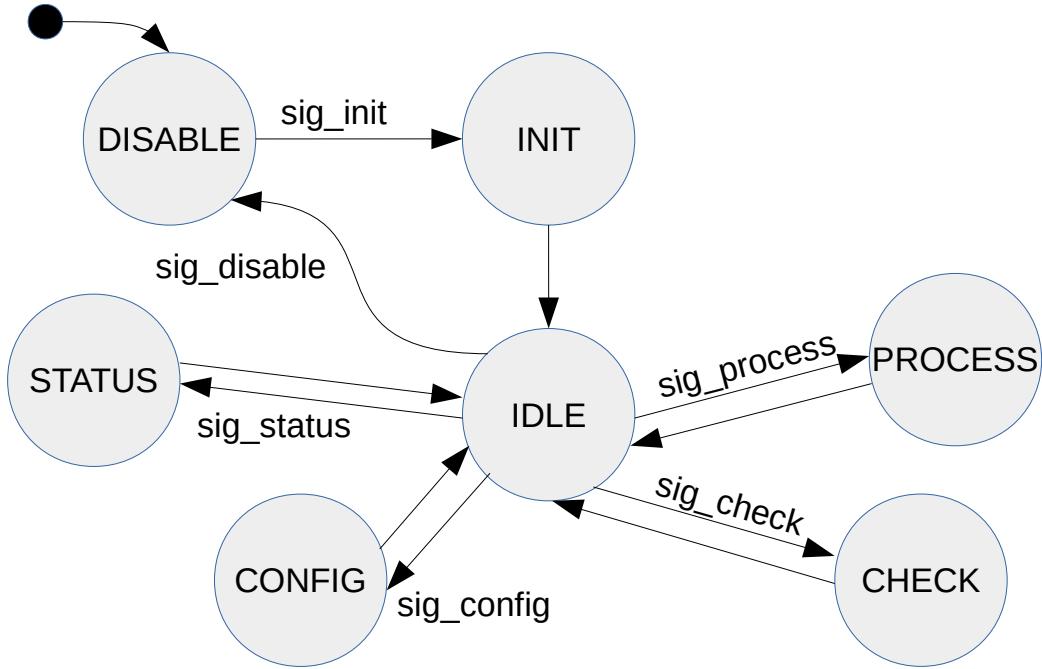


FIGURA 3.13: Máquina de estados finitos principal del módulo de interfaz máquina-hombre

TABLA 3.8: Descripción de los estados de la MEF principal del módulo de interfaz máquina-hombre (HMI). Se indican las señales que puede recibir junto con las acciones más relevantes de cada estado.

Estado	Señal para cambio	Acciones y actividades
DISABLE	Reset o sig_disable	Desinicializar la UART. Deshabilitar la interrupción de la UART. Deshabilitar la transmisión de la UART Cambiar el estado del módulo de READY a DISABLE.
INIT	sig_init	Inicializar la UART. Habilitar la interrupción de la UART. Habilitar la transmisión de la UART. Cambiar el estado del módulo de DISABLE a READY.
IDLE	transición incondicional	Cambiar estado del módulo de PROCESSING a READY.
CONFIG	sig_config	Realizar cambios de configuración al módulo.
CHECK	sig_check	Realizar autochequeo e informar al módulo de control
PROCESS	sig_process	Procesar los comandos recibidos
STATUS	sig_status	Enviar información de un módulo al usuario

3.5. Módulo de control

El propósito del módulo de control es implementar un *framework* o marco de trabajo para el funcionamiento de los demás módulos del sistemas. En este sentido, provee los mecanismos de *scheduling* para planificar la ejecución de las tareas del sistema y un *dispatcher* para ejecutarlas.

Asimismo, se incluyen métodos para temporizar regularmente la ejecución de los módulos que requieran tareas periódicas mediante el uso de una base de tiempo con el *sysTick timer* del microcontrolador.

Asimismo, este módulo implementa los mecanismos de comunicación entre procesadores descriptos en la subsección 2.4.1 con algunas modificaciones.

En primer lugar se definieron dos colas de mensajes idénticas, una para cada procesador, como puede verse en el algoritmo 3.3. Cada cola permite alojar una cantidad `IPCEX_QUEUE_SZ` de mensajes definida mediante una macro con valor numérico igual a 64.

```
/* Message QUEUE used by IPC library to exchange
   message between cores */
static ipcex_msg_t ipcex_coreM4[IPCEX_QUEUE_SZ];
static ipcex_msg_t ipcex_coreM0[IPCEX_QUEUE_SZ];
```

ALGORITMO 3.3: Definición de dos colas de eventos, una para cada procesador.

En segundo lugar, se permitió que los procesadores puedan escribir indistintamente en ambas colas, con el resguardo de activar la interrupción cruzada únicamente cuando el mensaje a encolar fuera para el otro procesador. Esta modificación es sencilla de implementar ya que la información del procesador de destino está contenida en el mismo mensaje.

De esta manera se proveyó un mecanismo de comunicación entre módulos definidos en un mismo procesador. Asimismo, esto otorgó mayor flexibilidad al sistema en su conjunto, ya que a partir de esta modificación, el código de un módulo pueden incluirse en ambos procesadores y mediante una macro, elegir desde qué procesador se va a ejecutar, sin otras modificaciones en el resto del código.

Por otra parte, se modificó el tipo de dato `ipcex_t` para contener, en lugar de dos campos de datos `uint32_t`, un campo tipo `signal_t` y otro tipo (`void *`) para el pasaje de parámetros genéricos, como puede verse en el algoritmo 3.4.

```
typedef struct __ipcex_msg {
    struct {
        cpuID_t cpu;
        moduleID_t pid;
    } id;
    signal_t signal;
    void * param;
} ipcex_msg_t;
```

ALGORITMO 3.4: Definición nuevo tipo de dato `ipcex_msg_t` para intercambio de mensajes.

El tipo de dato `signal_t` es una redefinición de una enumeración que contiene a todas las señales que es posible enviar a los módulos. Su definición se puede apreciar en el algoritmo 3.5. Se define una última señal `sig_invalid` para control de errores.

```
typedef enum {
    sig_disable,
    sig_init,
    sig_config,
    sig_check,
    sig_timeout,
    sig_process,
    sig_status,
    sig_acquire,
    sig_write,
    sig_read,
    sig_invalid
} signal_t;
```

ALGORITMO 3.5: Definición nuevo tipo de dato `signal_t` para el envío de señales a los módulos.

Se obtiene acceso para controlar los distintos módulos que componen el sistema mediante el registro de un puntero a la estructura de control tipo `module_t` de cada módulo, en un vector definido para tal fin. Dicho vector se indexa mediante un índice tipo `moduleID_t` cuya definición puede observarse en el algoritmo 3.6. Se define un último índice `INVALID_MODULE` para control de errores.

```
typedef enum {
    ONEWIRE,
    HMI,
    SDCARD,
    CONTROL,
    INVALID_MODULE
} moduleID_t;
```

ALGORITMO 3.6: Definición nuevo tipo de dato `moduleID_t` para indexar el control de los módulos.

Las distintas estructuras de control son variables privadas definidas en archivos propios de cada módulo. Para acceder a la dirección de memoria que las contiene, cada módulo ofrece una interfaz `x_get_module_pointer()` que devuelve por valor un puntero a `module_t`. La 'x' debe reemplazarse por el nombre del módulo correspondiente. Se exemplifica su uso en el algoritmo 3.7.

```
void FR_register_all_modules(void) {
    modules[ONEWIRE] = ow_get_module_pointer();
    modules[HMI] = hmi_get_module_pointer();
    modules[SDCARD] = sd_get_module_pointer();
    modules[CONTROL] = fr_get_module_pointer();
}
```

ALGORITMO 3.7: Función para registrar los módulos del sistema.

En el algoritmo 3.8 se presenta la función para despachar tareas y mensajes. Se puede observar cómo primero se recorren los módulos en busca del flag `runme` para ejecutar el manejador de eventos que tenga procesamiento pendiente. Se utiliza un mensaje especial vacío para este caso. Asimismo, se saca un mensaje de la cola si lo hubiere y se lo despacha al manejador de eventos correspondiente.

```

bool_t FR_dispatch_tasks (void) {

    moduleID_t moduleID;
    ipcex_msg_t msg;
    bool_t goToSleep;

    for (moduleID=0;moduleID<INVALID_MODULE;moduleID++) {
        if (modules [moduleID]→runme == TRUE)
            (*modules [moduleID]→eventHandler )(&msg_null);
    }

    if (IPC_tryPopMsg(&msg) == QUEUE_VALID) {
        if (msg . id . pid < INVALID_MODULE) {
            moduleID = msg . id . pid ;
            if (modules [moduleID]→status == DISABLED &&
                msg . signal == sig_init)
                (*modules [moduleID]→eventHandler )(&msg);

            else if (modules [moduleID]→status == READY)
                (*modules [moduleID]→eventHandler )(&msg);

            else
                IPC_tryPushMsg (msg . id . cpu , &msg);
        }
        else
            DEBUGSTR( "Error! msg . id . pid unknown\r\n");
    }

    goToSleep = TRUE;
    // Is there any module with processing pending?
    for (moduleID=0; moduleID<INVALID_MODULE; moduleID++)
    {
        if (modules [moduleID]→runme == TRUE)
            goToSleep = FALSE;
    }
    // Are there still messages in the queue?
    if ( IPC_msgPending(CPUID_CURR) != 0 )
        goToSleep = FALSE;

    return goToSleep;
};

```

ALGORITMO 3.8: Función para despachar tareas y mensajes.

Para el despacho de eventos se toman los recaudos de observar que se haya sacado un mensaje válido de la cola y que el módulo destinatario del mensaje esté correctamente identificado. A continuación, se consulta el estado del módulo. Si el módulo está deshabilitado y el mensaje contiene una señal `sig_init` se despacha el mensaje al módulo. Si el módulo se encuentra habilitado, se consulta si se encuentra en condiciones de recibir el mensaje (estado `READY`) en cuyo caso se despacha el mensaje al módulo. En caso contrario, si el módulo se encuentre habilitado pero en estado `PROCESSING`, el mensaje se vuelve a introducir en la cola a la espera que el módulo termine el procesamiento pendiente.

Finalmente, se observa si quedan módulos con procesamiento pendiente y si hay mensajes en la cola sin atender. Si alguna de estas condiciones resulta verdadera se deshabilita el flag `goToSleep` que se utiliza en el loop principal para saber si se debe poner el procesador en modo bajo consumo hasta que se produzca una interrupción.

En el algoritmo 3.9 se puede apreciar la estructura de *super loop* del punto de entrada al código luego de la secuencia de *startup*. Aquí se realizan las inicializaciones generales que no sean propias de los módulos, se registran los punteros a las estructuras de control y se envía una señal de inicialización a todos los módulos correctamente registrados. En el *loop* principal se utiliza la macro `__WFI()` para poner el procesador en modo bajo consumo hasta la siguiente interrupción según el estado del flag `goToSleep` como ya fuera mencionado.

```
int main(void)
{
    bool_t goToSleep = FALSE;

    prvSetupHardware();
    FR_register_all_modules();
    FR_broadcast_signal(sig_init);

    while(TRUE) { // the main loop

        goToSleep = FR_dispatch_tasks();

        if (goToSleep == TRUE)
            __WFI();
    }
    return 0;
}
```

ALGORITMO 3.9: Función principal `main()`.

Existe únicamente tres fuentes de interrupción posibles que pueden sacar al procesador del estado de bajo consumo, a saber:

- Interrupción del sysTick Timer que se usa como base de tiempo en ms.
- Interrupción de la UART cuando se recibe un comando para la interfaz con el usuario.
- Interrupción del otro procesador para indicar que ha enviado un mensaje que requiere procesamiento.

En las rutinas de servicio de cada una de estas interrupciones se encolan los mensajes correspondientes que serán debidamente atendidos en la siguiente ejecución del despachador de eventos.

En este módulo se implementa el control del perfil de consumo de energía del sistema, mediante lo cual se da cumplimiento al requerimiento 2.4.

Se definieron dos perfil de consumo distintos con las siguientes características:

- Perfil 1: optimizado para *performance*
 - Todos los módulos habilitados
 - Interrupción del sysTick Timer cada 1 ms
 - Las adquisiciones de temperatura y su escritura en la microSD cada 1 segundo.
- Perfil 2: optimizado para *consumo*
 - Módulo HMI deshabilitado
 - Interrupción del sysTick Timer cada 10 ms
 - Las adquisiciones de temperatura y su escritura en la microSD cada 10 segundo.

El sistema se inicia con la configuración del perfil 1 de consumo eléctrico y puede pasar al perfil 2 si el usuario envía una señal de configuración para tal fin a través de la interfaz HMI. A los fines de esta etapa del desarrollo, una vez en el perfil 2, sólo es posible volver al perfil 1 reseteando el microcontrolador, ya que la interfaz HMI dejará de estar disponible. Se contempla a futuro la posibilidad de intercalar ciclos de trabajo con los distintos perfiles según sea conveniente.

Por último, se menciona cuál fue la estrategia utilizada para asignar la ejecución de los módulos a los procesadores de la CIAA-NXP e integrar el sistema. Se descarga el mismo *firmware* a los procesadores cortex-M0 y cortex-M4 con todas las funcionalidades del sistema. Mediante un campo en la estructura de control de los módulos, `CPUID_T coreID`, se puede cambiar el procesador que ejecuta el módulo, tomando el recaudo de deshabilitar previamente el módulo en el procesador que lo estuviera ejecutando al momento del cambio.

La configuración inicial adoptada se puede apreciar en la tabla 3.9.

TABLA 3.9: Asignación de módulos a los procesadores

Procesador	Control	sdCard	Adquisición	HMI
Cortex-M0				X
Cortex-M4	X	X	X	

De esta forma, el procesador cortex-M0 permanecerá en modo bajo consumo la mayor parte del tiempo y sólo se activará cuando se produzca una interrupción. Las fuentes de interrupción posibles para este procesador en la configuración propuesta son la UART, cuando se recibe un comando por la interfaz de usuario y la interrupción de *software* del otro procesador, que se produce cuando un módulo ejecutándose en éste tenga un mensaje para sacar por la consola, mediante los mecanismos explicado en la sección 2.4.1.

Capítulo 4

Ensayos y Resultados

En este capítulo se describe la estrategia general de *testing* del proyecto y se documentan los ensayos realizados en tres niveles de abstracción: nivel de función, nivel de módulo y nivel de sistema.

4.1. Test Master Plan

Para la elaboración de un *Test Master Plan* se tomaron elementos del paradigma de desarrollo basado en las pruebas o TDD (por sus siglas en inglés, *Test Driven Development*) como fue mencionado en el capítulo 2, subsección 2.3.2.

Cabe destacar en este sentido, que en la planificación de las tareas del proyecto, primero se definieron los casos de prueba; luego se prepararon las herramientas de *testing*; luego se implementaron los módulos y finalmente, se realizaron las pruebas. Esto puede observarse en el diagrama de *Activity on Node* de la figura 2.1 en la sección 2.2.

Resultó necesario tener bien definidos los requisitos funcionales y sus criterios de aceptación. Se debieron contemplar todos los casos posibles de uso, tanto exitosos como de error, con lo que se elaboró un documento [20] de casos de prueba por módulo y de esta manera, se dio cumplimiento al requerimiento 1.2 indicado en la sección 2.1.

Se ensayó el código en tres niveles distintos de abstracción: a nivel de función, de módulo y de sistema, según se documenta en las subsecciones 4.1.1 Pruebas unitarias, 4.1.2 Pruebas funcionales y 4.1.3 Pruebas de sistema, respectivamente.

Asimismo, las pruebas sobre los módulos fueron diseñadas para evaluar únicamente la lógica principal de funcionamiento. En cada caso, se debió abstraer al módulo bajo ensayo de otras capas o servicios que pudieran interactuar con su lógica principal, para lo cual se simuló el resultado de dichas interacciones con *mocks*.

4.1.1. Pruebas unitarias

Para realizar pruebas unitarias al código se utilizó el framework Ceedling [21] que se distribuye como una gema del lenguaje de programación ruby. *Ceedling* integra tres herramientas, *Unity*, *Cmock* y *CException* y se encarga de compilar y ejecutar los *test* sobre el código.

A continuación se presentan los casos de prueba que se definieron para ensayar con *test* unitarios, las funciones de los cuatro módulos desarrollados.

Se utilizó una técnica para expandir la cantidad de *tests* posibles que consiste en desdoblar el ensayo sobre cada característica a evaluar en al menos tres *tests*: un *test* positivo, un *test* negativo y otro de rango cuando esto último fuera pertinente. Esto significa evaluar cuando una condición se cumple (positivo) y también cómo reacciona el sistema cuando no se cumple (negativo). Los *test* de rango permiten evaluar si los parámetros y resultados que devuelven las funciones ensayadas se encuentran dentro de los valores previsto y si se detecta cuando no lo están.

Módulo Almacenamiento

En la figura 4.1 se recopilan los casos de prueba para el módulo de almacenamiento. En la primer columna se incluye un código de identificación único del *test* que permite realizar la trazabilidad con los requerimientos.

Test ID	Tipo	Nombre del Test	Descripción
ALM_001	Positivo	Inicializar el FileSystem	Verificar la correcta inicialización del sistema de archivos
ALM_002	Positivo	Listar archivos presentes en la memoria SD	Verificar que se pueden listar correctamente los archivos presentes en la SD
ALM_003	Positivo	Leer estado vacío de la memoria SD	Verificar que se obtiene estado vacío cuando no hay archivos presentes
ALM_004	Positivo	Leer estado lleno de la memoria SD	Verificar que se obtiene estado lleno cuando no hay lugar para más archivos en la SD
ALM_005	Positivo	Abrir un archivo en la memoria SD	Verificar que se obtiene un puntero válido al archivo en la operación f_open()
ALM_006	Positivo	Leer un archivo en la memoria SD	Verificar que se lee correctamente el contenido de un archivo de texto plano
ALM_007	Positivo	Escribir un archivo en la memoria SD	Verificar que se puede escribir en un archivo en la SD
ALM_008	Positivo	Leer el estado del módulo de almacenamiento	Leer en qué estado se encuentra el controlador de almacenamiento
ALM_009	Positivo	Desactivar el módulo de almacenamiento	Pasivar el controlador de almacenamiento
ALM_010	Positivo	Activar el módulo de almacenamiento	Activar el controlador de almacenamiento
ALM_011	Negativo	No inicializar el FileSystem	Test ADQ_001 con inyección de falla mediante un función mock
ALM_012	Negativo	No listar archivos presentes en la memoria SD	Test ADQ_002 con inyección de falla mediante un función mock
ALM_013	Negativo	No leer estado vacío de la memoria SD	Test ADQ_003 con inyección de falla mediante un función mock
ALM_014	Negativo	No leer estado lleno de la memoria SD	Test ADQ_004 con inyección de falla mediante un función mock
ALM_015	Negativo	No abrir un archivo en la memoria SD	Test ADQ_005 con inyección de falla mediante un función mock
ALM_016	Negativo	No leer un archivo en la memoria SD	Test ADQ_006 con inyección de falla mediante un función mock
ALM_017	Negativo	No escribir un archivo en la memoria SD	Test ADQ_007 con inyección de falla mediante un función mock
ALM_018	Negativo	No leer el estado del módulo de almacenamiento	Test ADQ_008 con inyección de falla mediante un función mock
ALM_019	Negativo	No desactivar el módulo de almacenamiento	Test ADQ_009 con inyección de falla mediante un función mock
ALM_020	Negativo	No activar el módulo de almacenamiento	Test ADQ_010 con inyección de falla mediante un función mock

FIGURA 4.1: Casos de prueba para *test* unitarios del módulo de almacenamiento

Los *test* negativos se resolvieron con funciones *mock* que simulan el comportamiento erróneo del componente bajo prueba. No se realizaron *test* de rango para este módulo.

Todos los *test* ejecutados resultaron exitosos.

Módulo adquisición

Los casos de prueba para el módulo de adquisición se presentan en la figura 4.2. Para este módulo se definieron diez *test* de rango además de los *test* positivos y negativos. Para éstos últimos se utilizaron funciones *mock* para simular una falla.

Test ID	Tipo	Nombre del Test	Descripción
ADQ_001	Positivo	Identificar cada ds18b20 conectado al bus 1-wire	Identificar el ID único de cada sensor detectado en el bus 1-wire
ADQ_002	Positivo	Detectar presencia de cada ds18b20 identificado	Detectar la presencia de cada sensores de temperatura conectado al bus 1-wire
ADQ_003	Positivo	Adquirir valores de temperatura	Adquirir un valor de temperatura por cada sensor conectado al bus 1-wire
ADQ_004	Positivo	Adquirir periódicamente temperatura	Adquirir periódicamente un valor de temperatura de cada sensor conectado al bus 1-wire.
ADQ_005	Positivo	Leer configuración de ds18b20	Leer el contenido de la memoria interna de cada sensor de temperatura conectado al bus 1-wire
ADQ_006	Positivo	Escribir configuración de ds18b20	Escribir en la memoria interna de cada sensor de temperatura conectado al bus 1-wire
ADQ_007	Positivo	Convertir temperatura a celcius	Convertir el valor leído de cada sensor de temperatura de complemento a 2 a decimal.
ADQ_008	Positivo	Leer el estado del controlador de bus 1-wire	Leer en qué estado se encuentra el controlador del bus 1-wire
ADQ_009	Positivo	Desactivar el controlador del Bus 1-wire	Pasivar el controlador de bus 1-wire
ADQ_010	Positivo	Activar el controlador del Bus 1-wire	Activar el controlador de bus 1-wire
ADQ_011	Negativo	No identificar los ds18b20 conectados al bus 1-wire	Test ADQ_001 con inyección de falla mediante un función mock
ADQ_012	Negativo	No detectar presencia de ds18b20 identificados	Test ADQ_002 con inyección de falla mediante un función mock
ADQ_013	Negativo	No adquirir valores de temperatura	Test ADQ_003 con inyección de falla mediante un función mock
ADQ_014	Negativo	No adquirir periódicamente temperatura	Test ADQ_004 con inyección de falla mediante un función mock
ADQ_015	Negativo	No leer configuración de ds18b20	Test ADQ_005 con inyección de falla mediante un función mock
ADQ_016	Negativo	No escribir configuración de ds18b20	Test ADQ_006 con inyección de falla mediante un función mock
ADQ_017	Negativo	No convertir temperatura a celcius	Test ADQ_007 con inyección de falla mediante un función mock
ADQ_018	Negativo	No leer el estado del controlador de bus 1-wire	Test ADQ_008 con inyección de falla mediante un función mock
ADQ_019	Negativo	No desactivar el controlador del Bus 1-wire	Test ADQ_009 con inyección de falla mediante un función mock
ADQ_020	Negativo	No activar el controlador del Bus 1-wire	Test ADQ_010 con inyección de falla mediante un función mock
ADQ_021	Rango	Rango válido de valor de temperatura	Verificar el rango de temperatura obtenido en el test ADQ_003
ADQ_022	Rango	Rango válido de período de adquisición	Verificar que se acepta un valor de período con rango válido en el test ADQ_004
ADQ_023	Rango	Rango válido de valores de configuración	Verificar que se aceptan valores de configuración con rango válido en el test ADQ_006
ADQ_024	Rango	Rango válido de temperatura en Celcius	Verificar que el valor de temperatura convertido a grados Celsius esté en el rango válido
ADQ_025	Rango	Rango válido de estado del controlador de bus	Verificar que el estado del controlador del bus 1-wire esté en el rango válido
ADQ_026	Rango	Rango inválido de valor de temperatura	Verificar que se detecta un valor de temperatura fuera de rango pasado mediante una función mock
ADQ_027	Rango	Rango inválido de período de adquisición	Verificar que se detecta un valor de período fuera de rango pasado mediante una función mock
ADQ_028	Rango	Rango inválido de valores de configuración	Verificar que se detectan valores de configuración fuera de rango pasados mediante una función mock
ADQ_029	Rango	Rango inválido de temperatura en Celcius	Verificar que se detecta un valor de temperatura fuera de rango pasado mediante una función mock
ADQ_030	Rango	Rango inválido de estado del controlador de bus	Verificar que se detecta un estado del controlador fuera de rango pasado mediante una función mock

FIGURA 4.2: Casos de prueba para *test* unitarios del módulo de adquisición

Todos los *test* ejecutados resultaron exitosos.

Módulo HMI

En la figura 4.3 se recopilan los casos de prueba para el módulo de interfaz máquina-hombre. En la primer columna se incluye un código de identificación único del *test* que permite realizar la trazabilidad con los requerimientos. Los *test* de tipo negativo se implementaron con funciones *mock* para inyectar una falla.

Test ID	Tipo	Nombre del Test	Descripción
HMI_001	Positivo	Obtener la posición del cursor en la pantalla	Obtener la posición X,Y de dónde se encuentra el cursor en la pantalla de la terminal serie
HMI_002	Positivo	Desplazar el cursor al cmdline	Desplazar el cursor a la posición X,Y de la línea de comandos
HMI_003	Positivo	Desplazar el cursor a la barra de estado	Desplazar el cursor a la posición X,Y de la barra de estado
HMI_004	Positivo	Limpiar la pantalla	Enviar los comandos para borrar la pantalla y desplazar el cursor a la posición home
HMI_005	Positivo	Imprimir el encabezado	Enviar a la UART un string constante con el encabezado
HMI_006	Positivo	Imprimir menú de opciones contextual	Enviar a la UART un puntero a un string válido pasado por referencia.
HMI_007	Positivo	Imprimir mensaje de error	Imprimir un mensaje de error en la línea de comandos cuando se ingrese una opción no válida.
HMI_008	Positivo	Imprimir un mensaje de estado	Imprimir un mensaje de estado en la barra de estado
HMI_009	Positivo	Estado inicial de la MEF menú	Verificar el estado inicial de la MEF que maneja menú
HMI_010	Positivo	Obtener el estado de la MEF menú	Obtener el estado actual de la MEF que maneja el menú
HMI_011	Negativo	No obtener la posición del cursor en la pantalla	Test ADQ_001 con inyección de falla mediante un función mock
HMI_012	Negativo	No desplazar el cursor al cmdline	Test ADQ_002 con inyección de falla mediante un función mock
HMI_013	Negativo	No desplazar el cursor a la barra de estado	Test ADQ_003 con inyección de falla mediante un función mock
HMI_014	Negativo	No limpiar la pantalla	Test ADQ_004 con inyección de falla mediante un función mock
HMI_015	Negativo	No imprimir el encabezado inválido	Test ADQ_005 con inyección de falla mediante un función mock
HMI_016	Negativo	No imprimir menú de opciones contextual inválido	Test ADQ_006 con inyección de falla mediante un función mock
HMI_017	Negativo	No imprimir mensaje de error inválido	Test ADQ_007 con inyección de falla mediante un función mock
HMI_018	Negativo	No imprimir un mensaje de estado inválido	Test ADQ_008 con inyección de falla mediante un función mock
HMI_019	Negativo	Estado inicial de la MEF menú inválido	Test ADQ_009 con inyección de falla mediante un función mock
HMI_020	Negativo	No obtener el estado de la MEF menú	Test ADQ_010 con inyección de falla mediante un función mock
HMI_021	Rango	Rango válido para posición X del cursor	Verificar que se acepta un valor de posición X con rango válido en la función que desplaza el cursor
HMI_022	Rango	Rango válido para posición Y del cursor	Verificar que se acepta un valor de posición Y con rango válido en la función que desplaza el cursor
HMI_023	Rango	Rango válido para estado de MEF menú	Verificar que el estado de la MEF menú esté en el rango válido definido en <code>mef_menu_state_t</code>
HMI_024	Rango	Rango inválido para posición X del cursor	Verificar que se detecta un valor de posición X del cursor fuera de rango pasado mediante una función mock
HMI_025	Rango	Rango inválido para posición Y del cursor	Verificar que se detecta un valor de posición Y del cursor fuera de rango pasado mediante una función mock
HMI_026	Rango	Rango inválido para estado de MEF menú	Verificar que se detecta un estado de la MEF menú fuera de rango

FIGURA 4.3: Casos de prueba para *test* unitarios del módulo de interfaz máquina-hombre

Para este módulo se definieron tres *test* de rango válido y tres *test* de rango inválido. Éstos verifican los valores de posición X e Y en la función que desplaza el cursor por la pantalla y el valor de la variable de estado de la MEF que controla el menú de opciones contextual, respectivamente.

Todos los *test* ejecutados resultaron exitosos.

Módulo Control

En la figura 4.4 se recopilan los casos de prueba para el módulo de control. En la primer columna se incluye un código de identificación único del *test* que permite realizar la trazabilidad con los requerimientos.

Test ID	Tipo	Nombre del Test	Descripción
CON_001	Positivo	Obtener el estado de un módulo	Obtener un valor válido de estado cuando pide el estado de un módulo válido
CON_002	Positivo	Construir un mensaje	Construir exitosamente un mensaje con todos los campos definidos para ipcx_msg_t
CON_003	Positivo	Enviar un mensaje	Insertar exitosamente un mensaje en una cola de mensajes
CON_004	Positivo	Registrar todos los módulos	Registrar exitosamente todos los módulos en un vector de módulos
CON_005	Positivo	Enviar una señal a todos los módulos	Encolar exitosamente un mismo mensaje para todos los módulos registrados en el vector de módulos
CON_006	Positivo	Obtener exitosamente el estado de la cola de mensajes vacía	Obtener el estado correcto cuando se consulta una cola de mensajes que no tiene ningún elemento
CON_007	Positivo	Obtener exitosamente el estado de la cola de mensajes llena	Obtener el estado correcto cuando se consulta una cola de mensajes que se encuentra llena
CON_008	Positivo	Registrar todos los módulos	Registrar exitosamente todos los módulos en un vector de módulos
CON_009	Positivo	Enviar una señal a todos los módulos	Encolar exitosamente un mismo mensaje para todos los módulos registrados en el vector de módulos
CON_010	Positivo	Obtener el handler de un módulo válido	Obtener un puntero válido al manejador de eventos de un módulo cuando se requiere despachar un evento
CON_011	Negativo	No obtener el estado de un módulo	Test ADQ_001 con inyección de falla mediante un función mock
CON_012	Negativo	No lograr construir un mensaje	Test ADQ_002 con inyección de falla mediante un función mock
CON_013	Negativo	No lograr enviar un mensaje	Test ADQ_003 con inyección de falla mediante un función mock
CON_014	Negativo	No lograr registrar todos los módulos	Test ADQ_004 con inyección de falla mediante un función mock
CON_015	Negativo	No lograr enviar una señal a todos los módulos	Test ADQ_005 con inyección de falla mediante un función mock
CON_016	Negativo	No obtener exitosamente el estado de la cola de mensajes vacía	Test ADQ_006 con inyección de falla mediante un función mock
CON_017	Negativo	No obtener exitosamente el estado de la cola de mensajes llena	Test ADQ_007 con inyección de falla mediante un función mock
CON_018	Negativo	No lograr registrar todos los módulos	Test ADQ_008 con inyección de falla mediante un función mock
CON_019	Negativo	No lograr enviar una señal a todos los módulos	Test ADQ_009 con inyección de falla mediante un función mock
CON_020	Negativo	No lograr obtener el handler de un módulo válido	Test ADQ_010 con inyección de falla mediante un función mock
CON_021	Rango	Rango válido para construir mensaje	Rango válido para todos los campos pasados como parámetros para construir un mensaje
CON_022	Rango	Rango válido para enviar mensaje	Rango válido para todos los campos pasados como parámetros para enviar un mensaje
CON_023	Rango	Rango válido de señal broadcast	Rango válido para la señal que se desea enviar a todos los módulos
CON_024	Rango	Rango inválido para construir mensaje	Detectar rango inválido en los campos pasados como parámetros para construir un mensaje
CON_025	Rango	Rango inválido para enviar mensaje	Detectar rango inválido en los campos pasados como parámetros para enviar un mensaje
CON_026	Rango	Rango inválido de señal broadcast	Detectar rango inválido en la señal que se desea enviar a todos los módulos

FIGURA 4.4: Casos de prueba para *test* unitarios del módulo de control

Para este módulo se definieron tres *test* de rango válido y tres *test* de rango inválido. Éstos verifican los valores pasados como parámetros a las funciones para construir y enviar mensajes, y el valor de la señal que se desea enviar a todos los módulos, respectivamente.

Todos los *test* ejecutados resultaron exitosos.

4.1.2. Pruebas funcionales

Se utilizaron pruebas funcionales para el módulo de almacenamiento desarrolladas por Santiago Germino para la sAPI [22]. En particular, se incluyeron en el código del proyecto las funciones `fatFsFunctionalTest` y `fatFsTest`. Se realizaron pequeñas modificaciones para adaptar su contenido a la CIAA-NXP, ya que las mismas fueron desarrolladas para ser ejecutadas en la EDU-CIAA-NXP.

La salida de las pruebas se produce por la terminal serie con el resultado que se aprecia en la figura 4.5.

```
-----
Bienvenido a la prueba de wrappers sdcard/usbms!
-----
Iniciando sdcard con configuracion:
    velocidad inicial 100000 Hz.
    velocidad de trabajo 25000000 Hz.
FSSDC: [InitSPI] New card status: Inserted.
FSSDC: [Init] Initialization begins.
FSSDC: [Init] New card status: Native Mode.
FSSDC: [Init] New card status: Initializing.
FSSDC: [Init] New card status: Ready (Fast Clock).
Inicio de sdcard OK! Unidad FatFs 'SDC:'.
NO se probara usbms.
Logueando STATUS de dispositivos...
STATUS: Tarjeta SD lista y montada.

-----
TEST sobre archivo 'SDC:/TEST.TXT'.
-----
TEST: Ejecutando 'f_open( WRITE )'...OK!
TEST: Ejecutando 'f_putc'...OK!
TEST: Ejecutando 'f_puts'...OK!
TEST: Ejecutando 'f_open( READ )'...OK!
TEST: Ejecutando 'f_read'...OK!

>>> INICIO CONTENIDO DEL ARCHIVO LEIDO >>>
La unidad bajo prueba es 'SDC:'
Lista de caracteres ASCII:
!"#$%&' ()*+,./0123456789:;<=>?@ABCDEFGHIJKLMNPQRSTUVWXYZ
[\]^_`abcdefghijklmnoprstuvwxyz{|}~
Fecha y hora de compilacion: Nov 22 2018 18:59:47
Estado de Salidas Digitales D00 a D03 en la prueba: 1111
<<< FIN CONTENIDO DEL ARCHIVO LEIDO <<<
-----
TEST OK!
-----
**FIN**
```

FIGURA 4.5: Salida por consola del test funcional para el módulo de almacenamiento

En el algoritmo 4.1 se incluye la versión adaptada de la función original que implementa los *test* funcionales y cuya salida fue mostrada en la figura 4.5.

```

static bool fatFsTest( const char *unidad )
{
    char buf[1024];
    char filename[64];
    FIL file;
    FRESULT fr;
    int r;

    sprintf( filename, "%s/TEST.TXT", unidad );

    uartWriteString( UART_USB, "\r\n" );
    sprintf( buf, "TEST sobre archivo '%s'.\r\n",
             filename );
    uartWriteString( UART_USB, buf );
    uartWriteString( UART_USB, "\r\n" );

    // Ver http://elm-chan.org/fsw/ff/00index_e.html
    // para una referencia de la API de FatFs

    // Abre un archivo. Si no existe lo crea, si existe,
    // lo sobreescribe.
    fatFsTestStart( "f_open( WRITE )" );
    fr = f_open( &file, filename, FA_CREATE_ALWAYS |
    FA_WRITE );
    if( fr != FR_OK )
    {
        fatFsTestERROR( fr );
        return false;
    }
    fatFsTestOK( );

    // Prueba de f_putc
    fatFsTestStart( "f_putc" );
    sprintf( buf, "La unidad bajo prueba es '%s'\r\n"
             "Lista de caracteres ASCII:\r\n", unidad );
    // Escribe mensaje
    for (uint32_t i = 0; i < strlen(buf); ++i) {
        r = f_putc( buf[i], &file );
        if (r < 1)
        {
            fatFsTestERROR( r );
            f_close( &file );
            return false;
        }
    }
}

```

```
// Escribe todos los caracteres UTF-8 que overlapean
// ASCII
// (sin combinaciones multibyte)
for (uint32_t i = 32; i < 127; ++i)
{
    r = f_putc( (TCHAR)i, &file );
    if (r < 1)
    {
        fatFsTestERROR( r );
        f_close( &file );
        return false;
    }
}
fatFsTestOK( );

// Prueba f_puts
fatFsTestStart( "f_puts" );
sprintf (buf, "\r\n"
         "Fecha y hora de compilacion del programa: %s %s
\r\n"
         "Estado de Salidas Digitales DO0 a DO3 en la
prueba: %i%i%i\r\n",
         __DATE__, __TIME__,
         gpioRead( DO0 ), gpioRead( DO1 ), gpioRead( DO2
), gpioRead( DO3 ));

r = f_puts( buf, &file );
if (r < 1)
{
    fatFsTestERROR( r );
    f_close( &file );
    return false;
}
fatFsTestOK( );

// Cierra el archivo y vuelve a abrirlo como LECTURA
f_close( &file );

fatFsTestStart( "f_open( READ )" );
fr = f_open( &file , filename , FA_READ );
if( fr != FR_OK )
{
    fatFsTestERROR( fr );
    return false;
}

fatFsTestOK( );
```

```

// Borro contenido del buffer, para que no haya
// dudas de que el contenido se leyó desde el
// archivo
memset( buf, 0, sizeof(buf) );

// Carga el contenido del archivo
UINT bytesLeidos = 0;
fatFsTestStart( "f_read" );
fr = f_read( &file, buf, sizeof(buf), &bytesLeidos )
;
if (fr != FR_OK)
{
    fatFsTestERROR( fr );
    return false;
}
fatFsTestOK( );

fclose( &file );

uartWriteString( UART_USB, "\r\n");
uartWriteString( UART_USB, ">>> INICIO CONTENIDO
DEL ARCHIVO LEIDO >>>\r\n");
uartWriteString( UART_USB, buf );
uartWriteString( UART_USB, "<<< FIN CONTENIDO DEL
ARCHIVO LEIDO <<<\r\n");
return true;
}

```

ALGORITMO 4.1: Código para las pruebas funcionales para el módulo de almacenamiento.

4.1.3. Pruebas de sistema

Las pruebas que se presentan en esta sección son las de mayor nivel de abstracción, es decir que evalúan el funcionamiento del código a nivel de sistema con todos los módulos ya integrados al *firmware* y en pleno funcionamiento.

Se permiten las interacciones reales entre los módulos y entre éstos y las distintas capas y servicios que componen el código. En otras palabras, no se utiliza ningún *mock* para estos ensayos.

Se utilizó el modelo de casos de uso para plantear escenarios representativos del funcionamiento esperado del sistema, de modo que pueda ser evaluado su correcto desarrollo. Se procuró que estos escenarios estuvieran fuertemente vinculados con los requerimientos funcionales planteados en la sección 2.1.

Al final de esta sección se incluye una matriz de trazabilidad entre los requerimientos funcionales y las pruebas de integración que se detallan en la presente sección.

Para la construcción de las pruebas a nivel de sistema se confeccionó una plantilla que se adjunta a la memoria en el apéndice A.

El primer caso de prueba, identificado con ID UC01, evalua el escenario principal de funcionamiento con la configuración por defecto con que inicia el sistema al ser energizado. Esto es, todos los módulos habilitados y adquisición de temperatura en forma periódica cada 1000 ms. En la figura 4.6 se puede apreciar la planilla utilizada para el ensayo.

PROYECTO: EAMMRA			
ID CASO DE PRUEBA:	UC01	Test diseñado por:	Patricio Bos
NOMBRE DEL CASO:	Medición de temperatura autónoma	Fecha de diseño:	01/10/18
Versión de firmware:	1.0	Test ejecutado por:	Patricio Bos
		Fecha de ejecución:	01/11/18
Pre condiciones:	2 sensores DS18B20 conectados al puerto 1-WIRE (GPIO3[0]) de la CIAA-NXP. 1 Tarjeta microSD con formato FAT32 conectada al puerto SPI de la CIAA-NXP. Terminal serie conectada en configuración 8N1 115200 a la UART-USB		
Post condiciones:	ÉXITO: Valores de temperatura y timestamp en archivo de registro en la microSD FALLA: Archivo de registro en la tarjeta de memoria microSD vacío.		
Resumen del Test:	Test de funcionamiento autónomo. Después de energizar el sistema, se inicializan los módulos y se comienza a registrar en forma periódica valores de temperatura junto con una marca de tiempo en la tarjeta de memoria microSD. Se debe esperar 1 minuto desde la inicialización del sistema para y consultar el contenido del archivo de registro desde la interfaz HMI.		
Step #	Descripción	Resultado Esperado	Resultado Obtenido
1	Energizar el sistema	Mensaje de startup en la consola	ok
2	(sistema) Inicialización de todos los módulos	Mensajes de inicialización de cada módulo en la consola	ok
3	(sistema) Expira el timeout para la tarea periódica de adquisición de temperatura. Se encola una señal sig_timeout para el módulo de adquisición	-	-
4	(sistema) El módulo de adquisición recibe la señal sig_timeout y realiza una adquisición de temperatura.	-	-
5	(sistema) El módulo de adquisición envía una señal sig_write al módulo de almacenamiento con el dato de temperatura medido	-	-
6	(sistema) El módulo de almacenamiento recibe la señal sig_write y un valor de temperatura para guardar. El módulo de almacenamiento adquiere una marca de tiempo del RTC del sistema. El módulo de almacenamiento escribe una entrada en la tarjeta microSD con el valor de temperatura y una marca de tiempo	-	-
7	Esperar que se cumpla 1 minuto desde el paso #1	-	-
8	En la consola, ingresar al modo debug	Visualizar en la consola el menú contextual del modo debug	ok
9	En la consola, ingresar a la opción ver archivo de registro	Visualizar entradas con valores de temperatura con saltos regulares en la marca de tiempo	ok

FIGURA 4.6: Planilla de caso de uso UC01. Medición autónoma de temperatura en forma periódica. Se indica en color verde la post condición de éxito del ensayo.

Se pudieron validar todos los pasos y el ensayo resultó exitoso.

El segundo caso de prueba, identificado con ID UC02 evalua un escenario en donde el usuario interactua con el sistema para modificar el período de adquisición de valores de temperatura. Los detalles del ensayo y los resultados obtenidos se presentan en la planilla que se utilizó para la prueba, que se muestra en la figura 4.7.

PROYECTO: EAMMRA			
ID CASO DE PRUEBA:	UC02	Test diseñado por:	Patricio Bos
NOMBRE DEL CASO:	Cambio de período de adquisición de temp.	Fecha de diseño:	01/10/18
Versión de firmware:	1.0	Test ejecutado por:	Patricio Bos
		Fecha de ejecución:	01/11/18
Pre condiciones:	2 sensores DS18B20 conectados al puerto 1-WIRE (GPIO3[0]) de la CIAA-NXP. 1 Tarjeta microSD con formato FAT32 conectada al puerto SPI de la CIAA-NXP. Terminal serie conectada en configuración 8N1 115200 a la UART-USB		
Post condiciones:	ÉXITO: Cambio en el intervalo entre marcadas de tiempo de las mediciones. FALLA: Sin cambio en el intervalo entre marcadas de tiempo de las mediciones.		
Resumen del Test:	Test de cambio de configuración en el período de adquisición de valores de temperatura. Después de energizar el sistema, se inicializan los módulos y se comienza a registrar en forma periódica valores de temperatura. Al ingresar un nuevo valor para el período de adquisición se debe observar un cambio proporcional entre las marcas de tiempo de los datos registrados en la tarjeta microSD.		
Step #	Descripción	Resultado Esperado	Resultado Obtenido
1	Energizar el sistema	Mensaje de startup en la consola	ok
2	(sistema) Inicialización de todos los módulos	Mensajes de inicialización de cada módulo en la consola	ok
3	Esperar a que se cumpla 1 minuto desde el paso #1	-	-
4	En la consola, ingresar al modo configuración. Luego ingresar al módulo de adquisición 1-WIRE	Visualizar en la consola el menú contextual del configuración del módulo de adquisición	ok
5	En la consola, ingresar a la opción cambiar configuración del archivo de registro. Luego ingresar 2000 ms	Visualizar entradas con valores de temperatura con saltos regulares en la marca de tiempo	ok
6	(sistema) Se encola un mensaje para el módulo de adquisición con la señal sig_config y el nuevo valor para el período de adquisición.	-	-
7	(sistema) El módulo de adquisición recibe la señal sig_config y un nuevo valor de período. Cuando expira el nuevo timeout, el módulo de adquisición envía una señal sig_write con un valor de temperatura al módulo de almacenamiento.	-	-
8	En la consola, ingresar al modo debug	Visualizar en la consola el menú contextual del modo debug	ok
9	En la consola, ingresar a la opción ver archivo de registro	Visualizar entradas con valores de temperatura con saltos regulares en la marca de tiempo	ok

FIGURA 4.7: Planilla de casos de uso UC02. Cambio de período de adquisición de valores de temperatura. Se indica en color verde la post condición de éxito del ensayo.

Se pudieron validar todos los pasos y el ensayo resultó exitoso.

En la figura 4.7 se presenta los detalles del tercer caso de prueba, identificado con ID UC03. En este ensayo se evalua un escenario en donde el usuario interactua con el sistema para modificar el perfil de consumo de energía de la estación.

En caso de éxito, se observa un último mensaje a través de la interfaz que confirma el cambio de configuración y ésta deja de estar operativa.

PROYECTO: EAMMRA			
ID CASO DE PRUEBA:	UC03	Test diseñado por:	Patricio Bos
NOMBRE DEL CASO:	Cambio de perfil de consumo eléctrico	Fecha de diseño:	03/10/18
Versión de firmware:	1.0	Test ejecutado por:	Patricio Bos
		Fecha de ejecución:	01/11/18
Pre condiciones:	2 sensores DS18B20 conectados al puerto 1-WIRE (GPIO3[0]) de la CIAA-NXP. 1 Tarjeta microSD con formato FAT32 conectada al puerto SPI de la CIAA-NXP. Terminal serie conectada en configuración 8N1 115200 a la UART-USB		
Post condiciones:	ÉXITO: Nueva configuración de consumo aplicada. FALLA: Nueva configuración de consumo no aplicada.		
Resumen del Test:	Test de cambio de perfil de consumo. Después de energizar el sistema, se inicializan los módulos y la estación comienza a funcionar con el perfil de consumo optimizado para performance. Después de 1 minuto, el usuario ingresa a la interfaz en modo configuración y cambia el perfil de consumo en las opciones del módulo de control. El sistema debe enviar un mensaje de confirmación a través de la interfaz.		
Step #	Descripción	Resultado Esperado	Resultado Obtenido
1	Energizar el sistema	Mensaje de startup en la consola	ok
2	(sistema) Inicialización de todos los módulos	Mensajes de inicialización de cada módulo en la consola	ok
3	Esperar que se cumpla 1 minuto desde el paso #1	-	-
4	En la consola, ingresar al modo configuración. Luego ingresar al módulo de control	Visualizar en la consola el menú contextual del configuración del módulo de control	ok
5	En la consola, ingresar a la opción cambiar configuración de perfil de consumo. Luego confirmar el cambio ingresando 's'.	Visualizar perfil de consumo actual y mensaje de confirmación antes de aplicar el cambio	ok
6	(sistema) Se encola un mensaje para el módulo de control con la señal sig_config y el nuevo valor para el perfil de consumo.	-	-
7	(sistema) El módulo de control recibe la señal sig_config y un nuevo valor de perfil de consumo. Se cambia el período del sysTickTimer de 1 ms a 10 ms. Se cambia el valor período en la estructura de control del módulo de almacenamiento 10 a 1. Se envían mensajes confirmando el cambio de configuración a través de la interfaz	Visualizar en la barra de estado de la consola un mensaje del módulo de control con el nuevo perfil de consumo aplicado.	ok
8	(sistema) El módulo de control encola un mensaje para el módulo HMI con la señal sig_disable. El módulo HMI recibe el mensaje y deshabilita la UART	La consola queda deshabilitada.	ok

FIGURA 4.8: Planilla de casos de uso UC03. Cambio de perfil de consumo de energía. Se indica en color verde la post condición de éxito del ensayo.

Se pudieron validar todos los pasos y el ensayo resultó exitoso.

En la figura 4.9 se puede observar el banco de medición utilizado para los ensayos a nivel de sistema. En la PC se encuentra en ejecución la interfaz de usuario. Pueden apreciarse el lector de tarjetas microSD a la derecha de la CIAA-NXP. Asimismo, dos termómetros digitales DS18B20 se encuentran conectados a través de un protoboard. Los sensores se encuentran sumergidos en vasos de precipitados con agua a distinta temperatura para facilitar su identificación en los registros.

Cuando resultó necesario realizar comprobaciones a las señales eléctricas de los dispositivos bajo ensayo, se utilizó un multímetro GwInsteek modelo GDM-396 y un osciloscopio digital Tektronix modelo MDO3052. Finalmente, se elaboró una

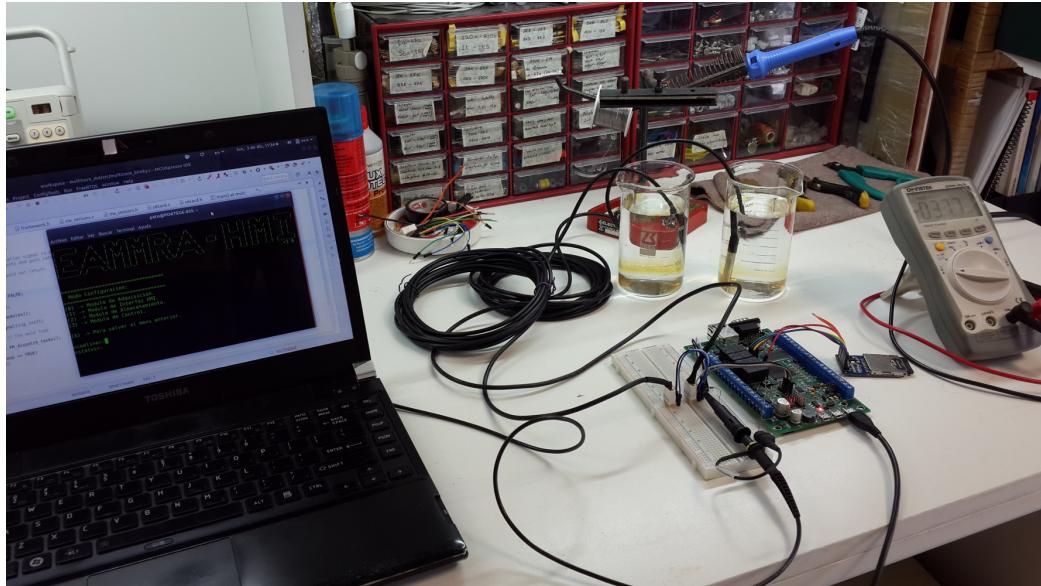


FIGURA 4.9: Banco de mediciones para los ensayos de sistema

matriz de trazabilidad de requerimientos con las pruebas de sistema para controlar fehacientemente que se pudieron alcanzar los requerimientos funcionales como se observa en la tabla 4.1.

TABLA 4.1: Matriz de trazabilidad de requerimientos con test de casos de uso.

Requerimiento	UC01	UC02	UC03
2.1 Adquirir temperatura	X	X	
2.2 Adquirir velocidad de viento			
2.3 Almacenar datos	X	X	
2.4 Perfiles de consumo			X
2.5 Interfaz	X	X	X

Cabe destacar que el requerimiento 2.2, referido a la adquisición de valores de velocidad de viento, no fue evaluado en ningún caso de uso. Esto responde a que dicho requerimiento no fue implementado. Cuando esta característica sea finalmente incluida en el sistema se contempla definir dos nuevos casos de uso que la utilice, uno para evaluar la adquisición periódica y otro para evaluar la posibilidad de cambiar dicho período.

Por último, en la tabla 4.1 puede apreciarse que todos los requerimientos implementados están alcanzados por al menos un caso de uso. Asimismo, debe notarse que si bien los casos de uso UC01 y UC02 cubren los mismos requerimientos, éstos prueban distintos aspectos del requerimiento 2.1. UC01 evalúa la adquisición periódica de valores de temperatura y UC02 la posibilidad de modificar dicho período de adquisición.

Capítulo 5

Conclusiones

En este capítulo se destacan los principales aportes del trabajo realizado y se listan los logros obtenidos. Asimismo, se documentan las técnicas que resultaron útiles para la ejecución del proyecto. Por otra parte, se deja constancia de las metas que no pudieron ser alcanzadas junto con las respectivas causas y se identifican las líneas de acción a futuro.

5.1. Conclusiones generales

En este trabajo se completó una primera iteración en el ciclo de diseño e implementación de un sistema embebido para el control de una estación de monitoreo de ruido ambiente submarino.

Se pudo desarrollar un *firmware* multicore modular que sienta las bases para una segunda iteración donde se diseñen e implementen tanto los componentes que quedaron fuera del alcance de esta etapa, así como también los que estaban contemplados pero no pudieron implementarse.

Los principales aportes de este trabajo son:

- Adopción de una metodología de desarrollo basada en control de versiones sistemática y robusta.
- Desarrollo de una arquitectura modular multicore que permite el intercambio de funcionalidades entre procesadores de forma flexible.
- Implementación de mecanismos de comunicación y sincronización entre procesadores, basados en interrupciones cruzadas y colas de mensajes.
- Implementación de mecanismos de control y despacho de tareas y eventos.
- Desarrollo de cuatro módulos con funcionalidad de adquisición de datos, almacenamientos de datos, HMI y control del sistema, respectivamente.
- Documentación completa de ingeniería de detalle de los módulos desarrollados.
- Documentación completa de testing.
- Documentación completa de trazabilidad entre requerimientos, ingeniería de detalle y testing.

5.1.1. Técnicas útiles

Resultaron particularmente útiles las técnicas de gestión de proyecto empleadas en la planificación de este trabajo. Mediante el desglose del proyecto en tareas y la articulación de éstas en diagramas de *Activity on Node* y *Gantt*, se pudo definir una metodología de seguimiento y control que permitió detectar retrasos en el avance del proyecto. Se aplicaron medidas de mitigación *ad-hoc* que consistieron en:

- el incremento de la dedicación horaria destinada al proyecto,
- el empleo de funciones *mock* o maquetadas para algunas características secundarias que pensaban implementarse.
- y en última instancia, la priorización del requerimiento implícito de fecha de finalización por sobre la cantidad de sensores operativos.

Dentro de la gestión de riesgos no se contempló explícitamente la posibilidad de imprevistos como los que efectivamente se manifestaron. Una análisis posterior de los hechos muestra que los retrasos responden principalmente a dos fuentes:

1. un evento fortuito, único e irrepetible, en el ámbito laboral, que sobrecargó la dedicación horaria a tareas ajena al proyecto durante gran parte de tiempo planificado para su realización e incluyó día de embarque.
2. la incorporación de nuevas responsabilidades en el campo de la docencia.

La primera fuente constituye un hecho aislado y, por su carácter de imprevisible, resulta razonable que haya quedado fuera del alcance de la gestión de riesgos.

La segunda fuente implica un error de cálculo en la relación entre horas disponibles y la carga horaria de las tareas de docencia asumidas. Esto último será tenido en cuenta como aprendizaje para futuros proyectos.

En términos generales, fue correcta la valoración de severidad y tasa de ocurrencia para todos los riesgos identificados; de los riesgos originalmente previstos:

1. No contar a tiempo con el anemómetro.
2. No contar con drivers para controlar el termómetro digital.
3. No contar con el becario PIDDEF a tiempo para que pueda participar en el proyecto.
4. Pérdida, robo o destrucción total o parcial de la placa CIAA-NXP y/o el anemómetro
5. Cancelación del proyecto PIDDEF del Ministerio de Defensa asociado.

sólo se manifestó íntegramente el riesgo número 3, que acertadamente fue valorado con una probabilidad de la ocurrencia alta. En este sentido, resultó adecuada la estrategia de planificar las tareas para ser llevadas a cabo sin la colaboración de un becario, por lo cual no se produjo ningún impacto negativo en el proyecto.

Respecto al riesgo número 5, si bien el proyecto no fue cancelado, el hecho de contar con un subsidio trianual como el otorgado por el programa PIDDEF [23], que haya coincidido con el cambio de gestión en la administración pública ocasionó retrasos y sobrecostos significativos que afortunadamente pudieron ser sobrelevados.

5.1.2. Metas alcanzadas

Se recopila en la tabla 5.1 el estado de los requerimientos, indicando mediante un tilde y color verde los que fueron alcanzados satisfactoriamente. Asimismo, se indica con una X y color rojo los requerimientos que no pudieron lograrse.

TABLA 5.1: Requerimientos alcanzados.

Requerimiento	Estado
1. Requerimientos de documentación	
1.1 Se debe generar un Memoria Técnica con la documentación de ingeniería detallada.	✓
1.2 Se debe generar un documento de casos de prueba.	✓
2. Requerimientos funcionales del sistema	
2.1 El sistema debe adquirir datos de un array de sensores de temperatura a intervalos regulares con un período de adquisición seleccionable.	✓
2.2 El sistema debe adquirir datos de un anemómetro a intervalos regulares con un período de adquisición seleccionable.	X
2.3 El sistema debe almacenar los datos de temperatura y velocidad de viento adquiridas junto con una marca de tiempo identificatoria en un medio físico no volátil.	✓
2.4 El sistema debe poder operar con dos perfiles de consumo de energía máximo desempeño y mínimo consumo de energía.	✓
2.5 El sistema debe contar con una interfaz serie cableada que permita realizar operaciones de configuración y mantenimiento.	✓
3. Requerimientos de verificación	
3.1 Se debe generar una matriz de trazabilidad entre la Memoria Técnica y los requerimientos.	✓
3.2 Se debe generar una matriz de trazabilidad entre las pruebas de integración y los requerimientos.	✓
4. Requerimientos de validación	
4.1 Se debe generar una matriz de trazabilidad entre el documento de casos de prueba y los requerimientos.	✓

A los fines prácticos de terminar el trabajo en la fecha pactada, no fue posible implementar el control del anemómetro. Esto no responde a una dificultad técnica sino a hechos imprevistos en la planificación que sobrecargaron la dedicación

horaria a actividades ajenas al proyecto e imposibilitaron el cumplimiento del requisito asociado a este sensor como fuera mencionado en la subsección 5.1.1.

5.2. Próximos pasos

Se indican en esta sección las líneas de acción más inmediatas para continuar la implementación del sistema e incorporar nuevas características a la estación de medición en desarrollo. Asimismo se establece un orden de prioridad para las acciones identificadas, a saber:

1. Implementar el control del anemómetro y completar los requerimientos previstos para esta etapa del proyecto.
2. Reemplazar las funciones *mock* que interactúan con la interfaz del sistema y así completar la implementación de todos los estados de las máquinas de estados finitos principales de cada módulo.
3. Incorporar más opciones de configuración a cada módulo. Para este fin, se contempla el pasaje por referencia de una estructura de configuración diferenciada para cada módulo que contenga los distintos parámetros configurables. Asimismo, se debe reformular el diseño de la interfaz máquina-hombre para permitir la toma de dichos parámetros.
4. Analizar la mejor forma de aprovechar las características asimétricas de los dos procesadores que posee la plataforma CIAA-NXP en cuanto a distribución de los módulos entre los procesadores. La arquitectura desarrollada permite que el mismo código sea descargado a ambos procesadores, debiendo elegirse en un archivo de configuración qué módulo se ejecuta en qué procesador para que no ocurran colisiones a la hora de acceder a los recursos. Esto permite explorar alternativas que aumenten la seguridad del sistema ante eventuales fallas de un procesador, existiendo redundancia de otro procesador que pueda retomar la ejecución de las tareas caídas. Asimismo, resulta de interés evaluar el desempeño del sistema en cuanto a consumo eléctrico, disipación térmica y tiempo de cómputo, entre otras características, en diferentes configuraciones de reparto de módulos entre los procesadores.
5. Incorporar al modelo de desarrollo un servidor de integración continua [24] para automatizar pruebas estáticas y dinámicas sobre el código.
6. Incorporar los subsistemas contemplados para futuras etapas en la figura 1.1 y que fueron considerados fuera del alcance en este proyecto, en forma de nuevos módulos de la estación, siguiendo el mismo patrón de diseño presentado en este trabajo.
7. Diseñar y fabricar un pocho para la CIAA-NXP que permita conectar y desconectar en forma robusta las diferentes entradas y salidas del sistema de forma de poder operar la estación en condiciones de campo.
8. Realizar pruebas de campo.

Apéndice A

Plantilla para casos de uso

PROYECTO: EAMMRA

ID CASO DE PRUEBA:

NOMBRE DEL CASO:

Versión de firmware:

Test diseñado por:

Fecha de diseño:

Test ejecutado por:

Fecha de ejecución:

Pre condiciones:

ÉXITO:

Post condiciones:

FALLA:

Resumen del Test:

Step #	Descripción	Resultado Esperado	Resultado Obtenido
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			

Bibliografía

- [1] Fred D Tappert. «The parabolic approximation method». En: *Wave propagation and underwater acoustics*. Springer, 1977, págs. 224-287.
- [2] H. Medwin y C.S. Clay. *Fundamentals of Acoustical Oceanography*. Applications of Modern Acoustics. Elsevier Science, 1997. ISBN: 9780080532165. URL:
<https://books.google.com.ar/books?id=kymUKicld2cC>.
- [3] R.J. Urick. *Principles of Underwater Sound*. McGraw-Hill, 1975. ISBN: 9780070660861. URL:
<https://books.google.com.ar/books?id=zAJRAAAAMA AJ>.
- [4] National Instruments. NI 6356 DEVICE SPECIFICATIONS.
<http://www.ni.com/pdf/manuals/374452c.pdf>. Visitada: 01-11-2018.
- [5] Patricio Bos. *Plan de Trabajo*. <http://laboratorios.fi.uba.ar/lse/tesis.html>. Visitada: 01-11-2018.
- [6] Vincent Driessens. *A successful Git branching model*.
<https://nvie.com/posts/a-successful-git-branching-model/>. Visitada: 01/11/2018.
- [7] K. Beck. *Test-driven Development: By Example*. Kent Beck signature book. Addison-Wesley, 2003. ISBN: 9780321146533. URL:
<https://books.google.com.ar/books?id=CUlsAQAAQBAJ>.
- [8] Robert C Martin. «Design principles and design patterns». En: () .
- [9] Adam Dunkels. *Protothreads, Lightweight Stackless Threads in C*.
<http://dunkels.com/adam/pt/>. Visitada: 01/11/2018.
- [10] Adam Dunkels y col. «Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems». En: *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*. Boulder, Colorado, USA, nov. de 2006. URL:
<http://dunkels.com/adam/dunkels06protothreads.pdf>.
- [11] Adam Dunkels, Oliver Schmidt y Thiemo Voigt. «Using Protothreads for Sensor Node Programming». En: *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*. Stockholm, Sweden, jun. de 2005. URL: <http://dunkels.com/adam/dunkels05using.pdf>.
- [12] MCUXpresso IDE User Guide. Rev. 10.2.0. NXP Semiconductors. 14 May 2018.
- [13] *LPC43xx/LPC43Sxx ARM Cortex-M4/M0 multi-core microcontroller - User manual*. UM10503. Rev. 2.3. NXP Semiconductors. 27 Jul 2017.
- [14] *Inter Processor Communication on LPC43xx*. AN1117. Rev. 2. NXP Semiconductors. 20 Ago 2014.
- [15] chaN. *FatFs - Generic FAT Filesystem Module*. Visitada: 01/11/2018. Electronic Lives Manufacturing. URL:
http://elm-chan.org/fsw/ff/00index_e.html.
- [16] The Linux Information Project. *BSD License Definition*.
<http://www.linfo.org/bsdlicense.html>. Visitada: 01-11-2018.

- [17] *DS18B20 Programmable Resolution 1-Wire Digital Thermometer.*
<https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf>. Rev. 5.
Maxim Integrated, Visitada: 01/11/2018.
- [18] James Harwood. *OneWire bus master implementation.*
https://community.nxp.com/servlet/JiveServlet/download/788225-1-387731/1100287_onewire-mk2.zip. Visitada: 01/11/2018.
- [19] Proyecto GNU. *GNU Screen*. <https://www.gnu.org/software/screen/>.
Visitada: 01-11-2018.
- [20] Patricio Bos. *Test Master Plan - Estación Autónoma de Monitoreo Marítimo de Ruido Acústico. Etapa I*. 2018.
- [21] ThrowTheSwitch.org. *Ceedling*.
<http://www.throwthewitch.org/ceedling/>. Visitada: 01-11-2018.
- [22] Santiago Germino. *Test funcional fatfs_sdCard*.
https://github.com/epernia/cese-edu-ciaa-template/blob/master/examples_c/sapi/fatfs_sdcard_usbmsd/src/fatfs_sdcard_usbmsd.c#L247.
Visitada: 01-11-2018.
- [23] Subsecretaría de Investigación Científica y Política Industrial para la Defensa. Ministerio de Defensa. *PROGRAMA DE INVESTIGACION Y DESARROLLO PARA LA DEFENSA*.
<https://www.argentina.gob.ar/defensa/piddef>. Visitada: 01-11-2018.
- [24] Martin Fowler. *Continuous Integration*.
<https://www.martinfowler.com/articles/continuousIntegration.html>.
Visitada: 01-11-2018.