UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE INGENIERÍA

MAESTRÍA EN SISTEMAS EMBEBIDOS

MEMORIA DEL TRABAJO FINAL

Sistema de control para estación autónoma de monitoreo de ruido ambiente submarino

Autor:
Esp. Ing. Patricio Bos

Director:
Dr. Ing. Ariel Lutenberg

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. Se utilizaron técnicas de programación multi-tarea cooperativa y se hizo uso extensivo de los conocimientos adquiridos durante la Maestría.

Agradecimientos

Agradecimientos personales. **[OPCIONAL]**

No olvidarse de agradecer al tutor.

No vale poner anti-agradecimientos (este trabajo fue posible a pesar de...)

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción General

En este capítulo se introduce brevemente el campo de la acústica submarina y la importancia del parámetro SONAR Ruido Ambiente Submarino como motivación para la realización de este trabajo. 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 $\sim$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 (***SO****und* ***N****avigation* ***A****nd* ***R****anging*).

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 (**SO**und **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 conveniente y lógicamente 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.

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 paquete tecnológico compuesto principalmente por 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.

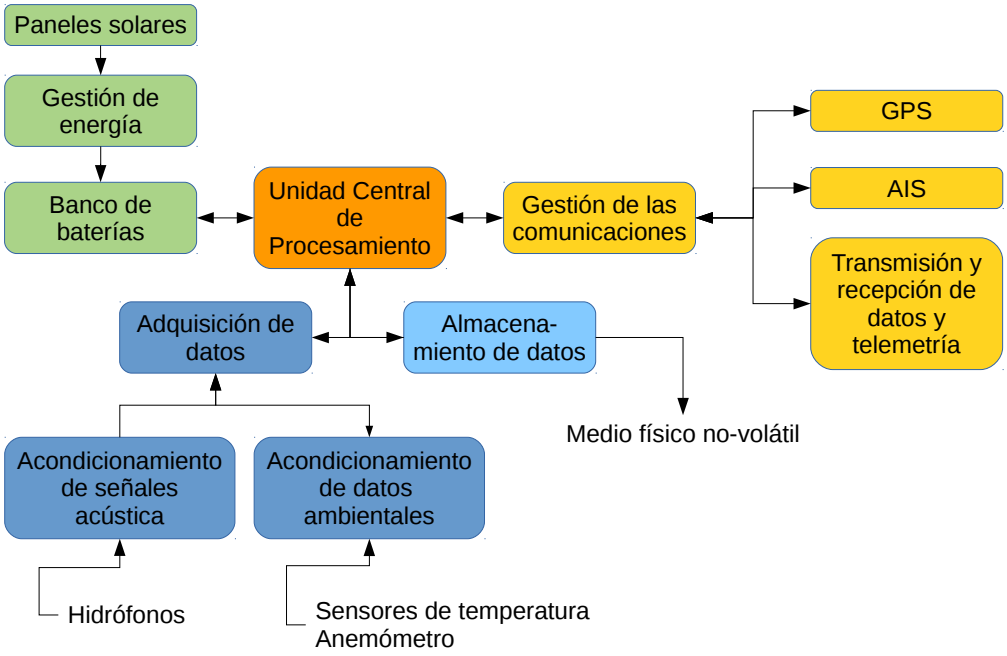


FIGURA 1.1: Diagrama en bloques general. Se diferencian por color los sub-módulos funcionales: energía; unidad central de procesamiento; comunicaciones; adquisición y almacenamiento.

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 boya. 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 excluídos 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.

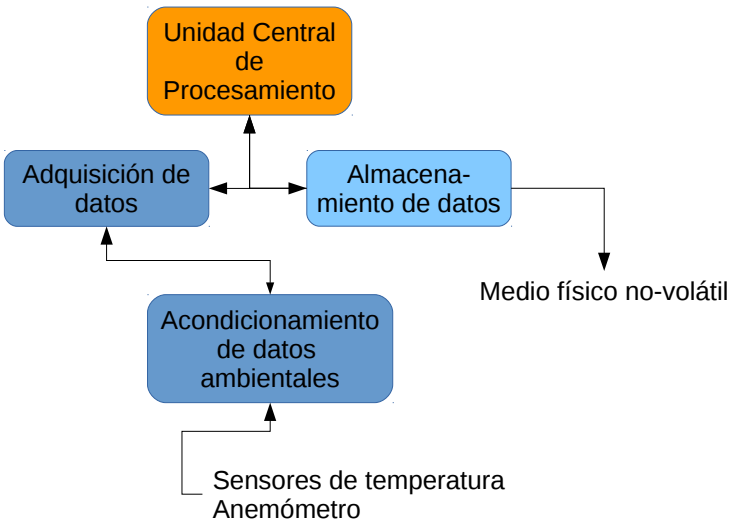


FIGURA 1.2: Diagrama en bloques del sistema implementado. Se diferencian por color los distintos sub-módulos funcionales incluidos en el alcance del proyecto.

Capítulo 2

Introducción Específica

En este capítulo se identifican aspectos relevante 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 [**insertar referencia**].

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

Se realizó un desglose de tareas que puede verse 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 detacan 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 será usado para realizar la trazabilidad de los requerimientos en las distintas etapas del proyecto.

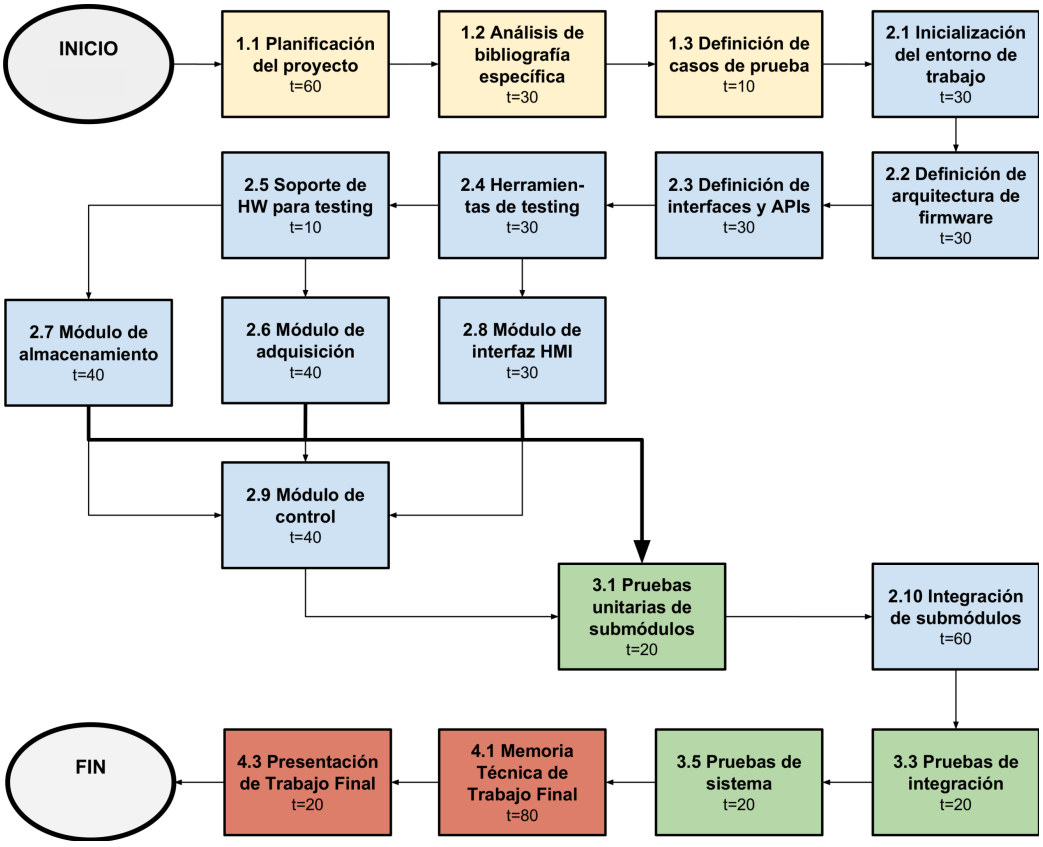


FIGURA 2.1: Diagrama *Activity on Node*. En colores puede distinguirse: en amarillo la etapa de documentación y análisis preliminar; en azul la etapa de diseño e implementación; en verde la etapa de verificación y validación y en rojo el proceso de cierre. El tiempo t está expresado en horas.

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"* [5]. 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.

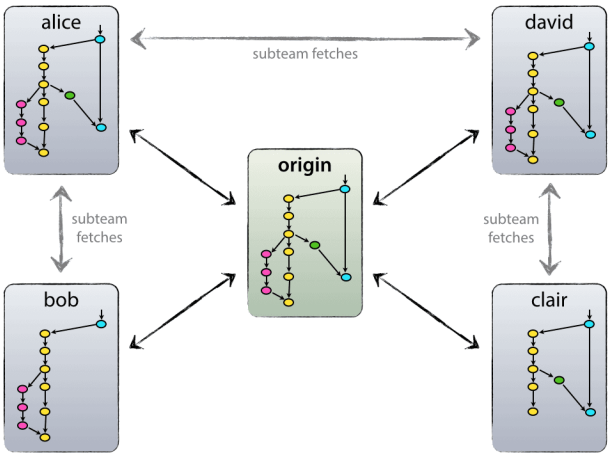


FIGURA 2.2: Esquema del flujo de trabajo entre repositorios[1].

Para la elaboración de este trabajo, donde la codificación recayó principalmente sobre una sóla 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

[1]Imagen 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.

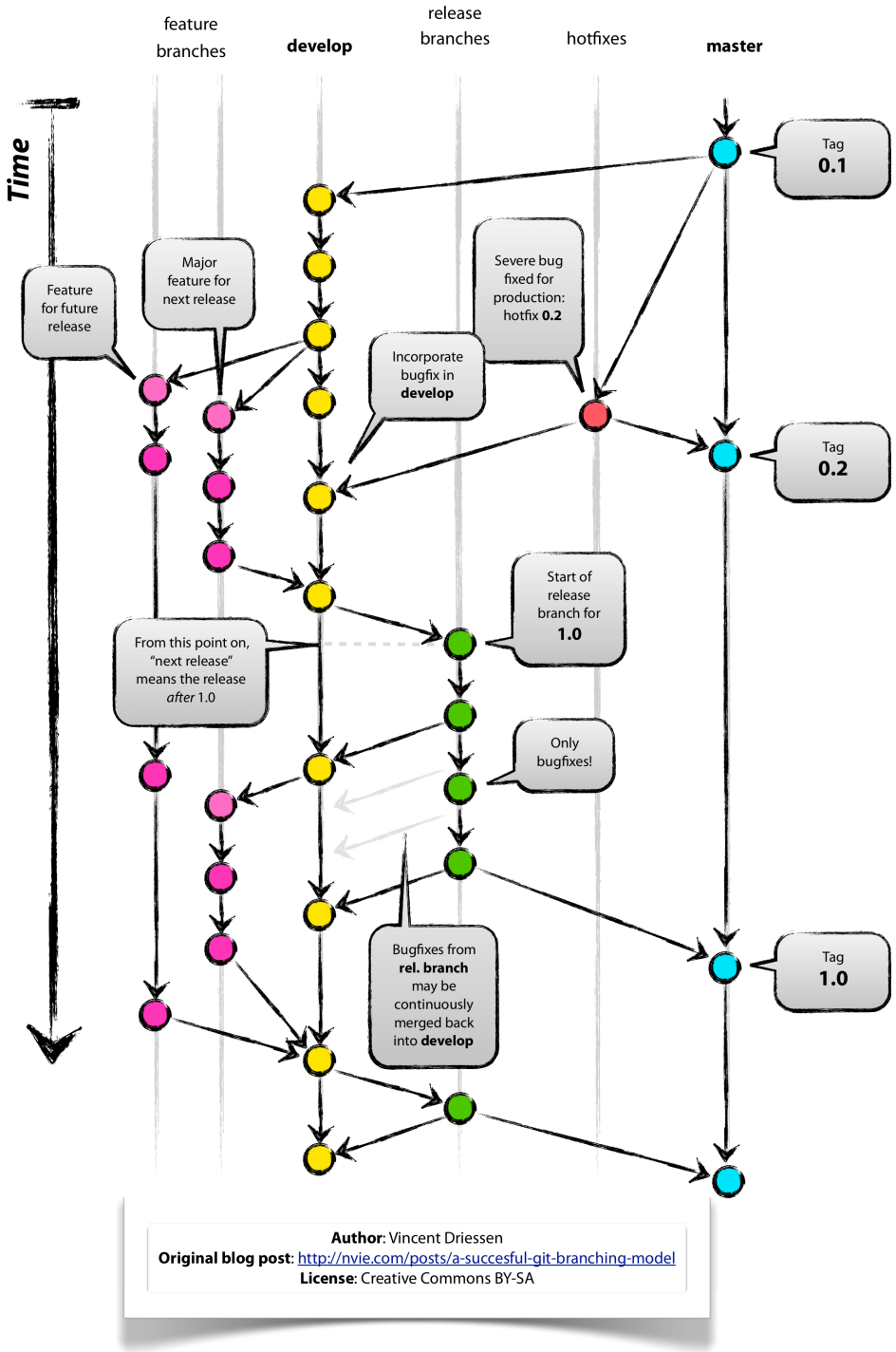


FIGURA 2.3: Modelo de ramas utilizado en git[2].

[2]Imagen 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. 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 [6].

Se distribuyen como una biblioteca que puede integrarse al proyecto y proveen la posibilidad de 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 [7] [8].

En el presente trabajo, se hace uso de protothreads en la codificación del protocolo de comunicación 1-wire que se describe en la subsección 3.4.1.

Capítulo 3

Diseño e Implementación

En este capítulo se presentan la arquitectura multicore del *firmware*, los mecanismos de comunicación entre procesadores y el detalle del diseño de los módulos desarrollados junto con sus interfaces. Finalmente, se incluye una sección de trazabilidad de requerimientos con las funciones implementadas.

3.1. Arquitectura multicore

El *firmware* está desarrollado sobre la base de dos proyectos vinculados del 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 [9]. 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 [10].

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 3.1 y 3.2 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 3.1: 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 3.2: 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 3.1.1.

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

3.1.1. 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" [11]. En este documento se explican 3 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.

Interrupción

El mecanismo de interrupciones cruzadas es el más simple de los 3 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.

```
/*
 * 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();
}
```

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 3.1 muestra esquemáticamente esta configuración.

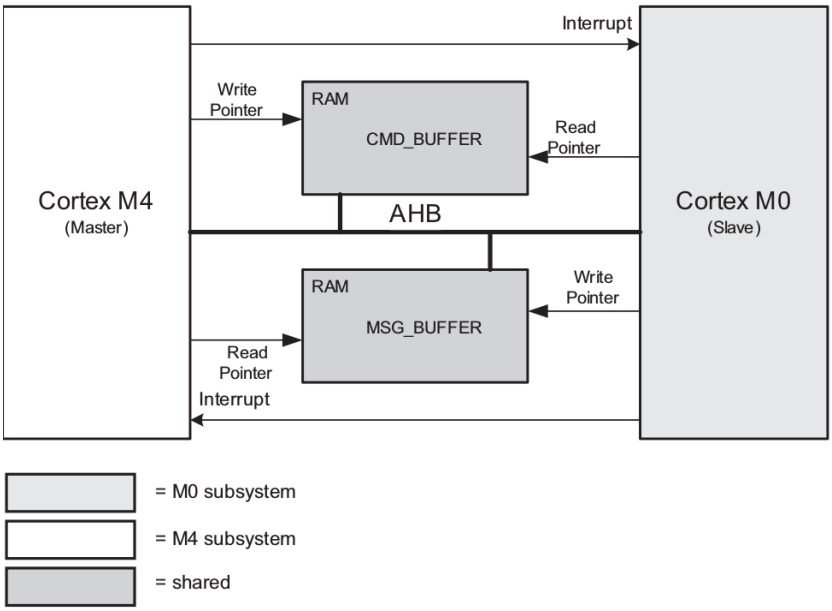


FIGURA 3.1: Esquema de comunicación entre procesadores basado en colas de mensajes[1].

[1]Imagen tomada del manual de usuario del microcontrolador LPC4337 [10]

En el esquema propuesto, sólamente el procesador maestro puede escribir en el búfer de comandos y recibe los mensaje 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 descrito en el apartado Interrupción, dentro de la subsección 3.1.1. Luego de esto, 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.

```
/* 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);
}
```

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.

Para los mensajes, se define un nuevo tipo de dato ipcex_msg_t como una estructura que contiene información que identifica al CPU y al proceso destinararios del mensaje junto con dos campos para datos.

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

  uint32_t data0;
  uint32_t data1;
} ipcex_msg_t;
```

3.2. Diseño de módulos y definición de interfaces

El *firmware* está fuertemente modularizado en archivos y utiliza un modelo de capas jerárquicas para organizar el código en distintos niveles de abstracción. En la figura 3.2 se pueden observar, agrupados 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 bibiotecas; 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.

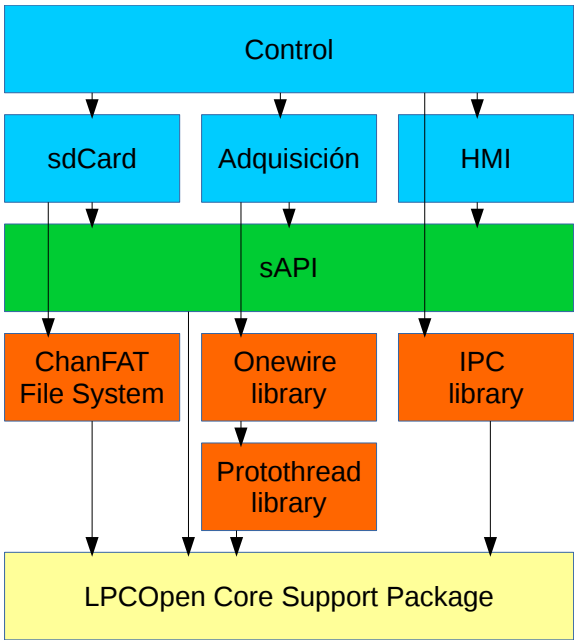


FIGURA 3.2: Estructura de capas para el *firmware*. En orden creciente de nivel de abstracción: *core support package* (amarillo), bibliotecas (naranja), sAPI (verde) y la capa de aplicación (celeste) con los 4 módulos implementados.

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 3.3 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 3.3: 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 una estructura de control para los módulos que incluye información sobre en qué *core* debe ejecutarse, un campo para identificar al módulo, un puntero a la función manejadora de eventos que es invocada cada vez que hay un mensaje para este módulo, un campo para temporizar la ejecución periódica del módulo y finalmente, 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;
```

Toda la interacción con los módulos se realiza a través de los respectivos manejadores de eventos y una cola de mensajes como la descrita en la subsección 3.1.1, pero definida en una posición de memoria no compartida, por lo que su uso se limita al *core* donde está definida.

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

- `void onewire_handler(const ipcex_msg_t * msg );`
- `void sdcard_handler(const ipcex_msg_t * msg );`
- `void hmi_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 enteros sin signo de 32 bits de datos que contiene el mensaje. Normalmente el campo `data0` define un comando y `data1` se utiliza opcionalmente para el envío de parámetros.

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 pueden o no estar modelados con MEFs dependiendo de la conveniencia.

3.3. 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*.

Se evaluaron distintas opciones de medios físicos. En la tabla 3.4 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.

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

TABLA 3.4: Alternativas de medios físicos evaluados

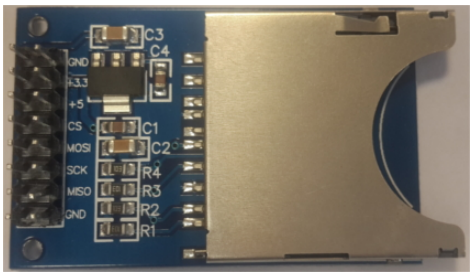
Medio físico	Capacidad	Interfaz	Protocolo
USB Mass Storage Device	∼10 Gb	USB 2.0	USB
Tarjeta de memoria	∼10 Gb	Micro SD	SSP
Disco de estado sólido (SSD)	∼100 Gb	SATA III	SATA
Disco duro mecánico (HHD)	∼1000 Gb	SATA III	SATA

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 entre la CIAA-NXP y el lector (figura 3.3a) y una vista superior del módulo de hardware (figura 3.3b).

CIAA-NXP		**Lector de tarjetas**
+3.3V	–>	+3.3V
GPIO0	–>	CS
SPI_MOSI	–>	MOSI
SPI_SCK	–>	SCK
SPI_MISO	–>	MISO
GND	–>	GND

(A)



(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 [12]. 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.

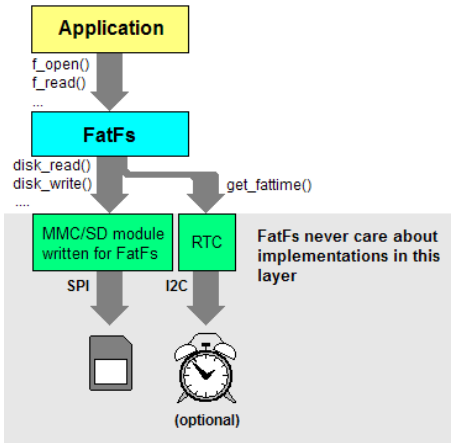


FIGURA 3.4: Diagrama de capas con la ubicación de la biblioteca fatFs en el sistema y sus interfaces[2].

[2]Imagen tomada de http://elm-chan.org/fsw/ff/doc/appnote.html.

El código está escrito en ANSI C y es *software* libre bajo licencia estilo BSD [13].

Se presenta el diagrama de estado de la MEF principal del módulo en la figura 3.5, donde puede apreciase el punto de entrada con un circulo negro, los estado 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 operación 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`.

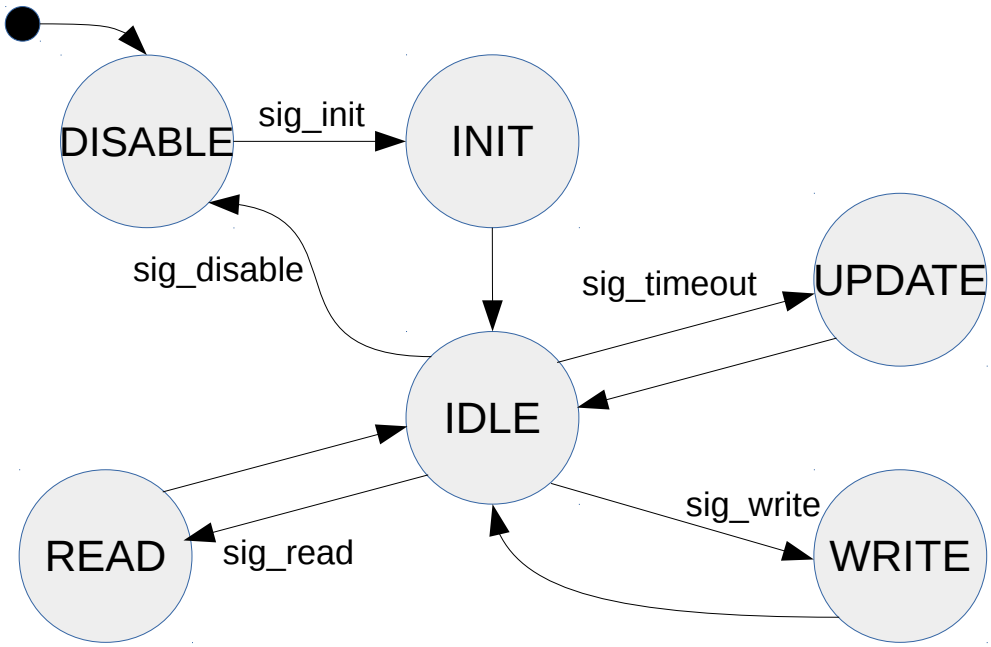


FIGURA 3.5: Máquina de estados finitos principal del módulo de almacenamiento sdCard

En la table 3.5 se describen cada uno de los posibles estados de la MEF principal del módulo de almacenamiento. Asimismo, se mencionan las señales para alcanzar cada uno de los estados de la MEF y se explicitan las acciones y actividades más destacables que se realizan en cada uno de ellos.

TABLA 3.5: Descripción de los estados de la MEF principal del módulo de almacenamiento.

Estado	Señal necesaria para alcanzarlo	Acciones y actividades
DISABLE	Reset o sig_disable	Desinicializar el controlador SPI. Desinicializar la bilioteca fatFS. Desinicializar el módulo de almacenamiento. Cambiar el estado del módulo de READY a DISABLE.
INIT	sig_init	Inicializar el controlador SPI. Inicializar la bilioteca fatFS. Inicializar el módulo de almacenamiento. Cambiar el estado del módulo de DISABLE a READY.
IDLE	Transición incodicional desde otros subestados al terminar su actividad.	Cambiar estado del módulo de PROCESSING a READY
UPDATE	sig_update	Ejecutar tarea periódica de mantenimiento de la tarjeta SD
WRITE	sig_write	Obtener timestamp Escribir datos en la tarjeta SD
READ	sig_read	Leer datos en la tarjeta SD

3.4. Módulo de adquisición

3.4.1. Sensor de temperatura

3.4.2. Sensor de velocidad de viento

3.5. Módulo interfaz HMI

3.6. Módulo de control

Capítulo 4

Ensayos y Resultados

4.1. Test Master Plan

4.1.1. Pruebas unitarias

4.1.2. Pruebas funcionales

4.1.3. Pruebas de sistema

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, de modo que sirva como experiencia para futuros proyectos. Finalmente, se identifican las líneas de acción a futuro y posibles mejoras para continuar el trabajo más adelante.

5.1. Conclusiones generales

La idea de esta sección es resaltar cuáles son los principales aportes del trabajo realizado y cómo se podría continuar. Debe ser especialmente breve y concisa. Es buena idea usar un listado para enumerar los logros obtenidos.

5.2. Próximos pasos

Acá se indica cómo se podría continuar el trabajo más adelante.

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=zAJRAAAAMAAJ.

[4] National Instruments. *NI 6356 DEVICE SPECIFICATIONS*. http://www.ni.com/pdf/manuals/374452c.pdf. Visitada: 01-11-2018.

[5] Vincent Driessen. *A successful Git branching model*. https://nvie.com/posts/a-successful-git-branching-model/. Visitada: 01/11/2018.

[6] Adam Dunkels. *Protothreads, Lightweight Stackless Threads in C*. http://dunkels.com/adam/pt/. Visitada: 01/11/2018.

[7] 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.

[8] 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.

[9] *MCUXpresso IDE User Guide*. Rev. 10.2.0. NXP Semiconductors. 14 May 2018.

[10] *LPC43xx/LPC43Sxx ARM Cortex-M4/M0 multi-core microcontroller - User manual*. UM10503. Rev. 2.3. NXP Semiconductors. 27 Jul 2017.

[11] *Inter Processor Communication on LPC43xx*. AN1117. Rev. 2. NXP Semiconductors. 20 Ago 2014.

[12] chaN. *FatFs - Generic FAT Filesystem Module*. Visitada: 01/11/2018. Electronic Lives Manufacturing. URL: http://elm-chan.org/fsw/ff/00index_e.html.

[13] The Linux Information Project. *BSD License Definition*. http://www.linfo.org/bsdlicense.html. Visitada: 01-11-2018.