Universidad Central "Marta Abreu" de Las Villas

Facultad de Ingeniería Eléctrica

Departamento de Automática y Sistemas Computacionales

TRABAJO DE DIPLOMA

Implementación de mejoras al software de nivel táctico de GARP para vehículos AUV.

Autor: Jorge Andrés Prado Giance

Tutor: Ing. Yeiniel Suárez Sosa

Santa Clara

2014

"Año 56 de la Revolución"

Universidad Central “Marta Abreu” de Las Villas

Facultad de Ingeniería Eléctrica

Departamento de Automática y Sistemas Computacionales

TRABAJO DE DIPLOMA

Implementación de mejoras al software de nivel táctico para el AUV del GARP

Autor: Jorge Andrés Prado Giance

e-mail: jprado@uclv.edu.cu

Tutor: Ing. Yeiniel Suárez Sosa

Dpto. de Automática, Facultad de Ing. Eléctrica, UCLV

e-mail: yeiniel@uclv.cu

Santa Clara

2014

"Año 56 de la Revolución"

Hago constar que el presente trabajo de diploma fue realizado en la Universidad Central "Marta Abreu" de Las Villas como parte de la culminación de estudios de la especialidad de Ingeniería en Automática, autorizando a que el mismo sea utilizado por la Institución, para los fines que estime conveniente, tanto de forma parcial como total y que además no podrá ser presentado en eventos, ni publicado sin autorización de la Universidad.

_____

Firma del Autor

Los abajo firmantes certificamos que el presente trabajo ha sido realizado según acuerdo de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____ _____

Firma del Autor

Firma del Jefe de Departamento
donde se defiende el trabajo

_____

Firma del Responsable de
Información Científico-Técnica

PENSAMIENTO

"*Un científico debe tomarse la libertad de plantear cualquier cuestión, de dudar de cualquier afirmación, de corregir errores*"

Robert Oppenheimer

"Una computadora puede ser llamada "inteligente" si logra engañar a una persona haciéndole creer que es un humano"

Alan Turing

DEDICATORIA

A mis padres Letissia y Jorge Andrés

Por darme vida y ejemplo

A mis tías Sila y Carmen

Por todos los recuerdos de mi infancia

AGRADECIMIENTOS

A mi familia.

A mis padres por apoyarme.

A mis tías que son también madres para mí.

A mis abuelos porque dijeron sí cuando otros decían no.

A mis hermanas por la experiencia que solo te puede dar un hermano.

A los que ya no están pero formaron parte de mi vida.

A Milena por comprenderme y soportarme más tiempo que nadie en estos 5 años.

A mi tutor Yeiniel por enseñarme y ser también mi amigo.

A mis amigos que siempre he podido contar con ellos Jose, Elie, Noel, los de Ceballo 2, los del parque Martí, los de Trabajadores Sociales y los del 304B del U2.

TAREA TÉCNICA

1. Realización de un estudio de los principales estándares de programación y los antecedentes en la implementación de software para AUV.
2. Caracterización del Software de nivel táctico para AUV de GARP.
3. Análisis de las funcionalidades necesarias para este tipo de software.
4. Implementación de mejoras que tributen al cumplimiento de los objetivos planteados.
5. Evaluación de la efectividad de la propuesta de mejoramiento.
6. Elaboración del informe de tesis.

Firma del Autor	Firma del Tutor

RESUMEN

El aprovechamiento a nivel comercial y el número de investigaciones basadas en vehículos autónomos sumergibles ha venido en crecimiento en los últimos años a nivel mundial. El desarrollo del software que da vida a estas estructuras robóticas viene adoptando una tendencia a la liberación del código fuente y el conocimiento que lo soporta. El Grupo de Automatización Robótica y Percepción (GARP) de la Universidad Central "Marta Abreu" de Las Villas (UCLV) en asociación con el Centro de Investigación y Desarrollo Naval (CIDNAV) desde hace algunos años viene desarrollando un sistema de supervisión y control para un prototipo de vehículo autónomo sumergible denominado HRC-AUV que presenta ciertos problemas de ejecución.

Este trabajo consiste en el perfeccionamiento del software de nivel táctico del HRC-AUV de GARP existente. La tarea principal es implementar un grupo de mejoras y funcionalidades que permitan el cumplimiento de los requisitos de tiempo real en el periodo de la tarea de adquisición de datos desde la Unidad de Mediciones Inerciales y eleven la calidad del proyecto. También se analizan los datos recopilados en experimentos reales y se describen los resultados obtenidos en las pruebas efectuadas a la aplicación.

TABLA DE CONTENIDOS

PENSAMIENTO iv

DEDICATORIA v

AGRADECIMIENTOS vi

TAREA TÉCNICA vii

RESUMEN viii

TABLA DE CONTENIDOS i

INTRODUCCIÓN 1

Descripción del entorno de la aplicación 2

Justificación 3

Situación del problema 3

Objetivo general 5

Objetivos específicos 5

Organización y estructura del trabajo de diploma 5

CAPÍTULO 1. ANTECEDENTES TEÓRICOS DEL DESARROLLO DE SOFTWARE PARA AUV…….. 7

1.1 Introducción 7

1.2 Arquitectura de software para robots 8

1.3 Ejemplos de AUVs 11

Phoenix AUV 11

ISiMI6000 11

Starfish 12

Mares 12

Twin-Burger....13
Yellowfin....14
1.4 Sistema Operativo....15
1.4.1 MetaOS....15
Robot Operating System (ROS)....16
Mission Oriented Operating Suite (MOOS)....16
Universal Robot Body Interface (URBI)....17
Player....17
1.5 Sistemas de Tiempo Real....17
1.5.1 Clasificaciones de los Sistemas de Tiempo Real....18
1.5.2 Programación....19
Hilos....19
API....20
1.6 Conclusiones parciales del capítulo....20
CAPÍTULO 2. HERRAMIENTAS Y TÉCNICAS DE IMPLEMENTACIÓN....21
2.1 Introducción....21
2.2 Entorno de ejecución....21
POSIX....22
2.3 Librerías para cálculo científico....24
2.4 Persistencia de Información de Depurado....25
2.5 Comportamiento de Demonio....27
2.6 Sistema de construcción....28
2.7 Analizadores de Rendimiento....31
2.8 Evaluación del software....34

2.9 Generación de Documentación 35
2.10 Conclusiones parciales del capítulo 37
CAPÍTULO 3. RESULTADOS Y ANÁLISIS 38
3.1 Introducción 38
3.2 Perfilado y requisitos de tiempo real 38
3.3 Análisis de *logs* 41
3.4 Evaluación del software 43
3.5 Documentación 43
3.6 Sistema de Construcción 45
3.7 Conclusiones parciales del capítulo 45
CONCLUSIONES Y RECOMENDACIONES 46
CONCLUSIONES 46
RECOMENDACIONES 47
REFERENCIAS BIBLIOGRÁFICAS 48
ANEXO A. ENCAPSULADO PARA LA FUNCIÓN PTHREAD_CREATE 51
ANEXO B. ANÁLISIS DE RENDIMIENTO 54
ANEXO C. DAEMON.C 55

INTRODUCCIÓN

Un Vehículo Autónomo Sumergible (AUV por sus siglas en inglés) es una estructura robótica con una fuente de energía propia y un sistema de cómputo capaz de procesar datos provenientes de sensores, comunicarse y ejecutar algoritmos de navegación que le permiten cierta autonomía en el cumplimiento de misiones. Forman parte de los vehículos submarinos teledirigidos (UUV por sus siglas en inglés) y se diferencian de los Vehículos Operados a Distancia (ROV por sus siglas en inglés) en que no se necesita la intervención humana para operarlos (Siciliano and Khatib, 2008).

El avance en la ciencia y la tecnología propician la exploración de zonas desconocidas. En la carrera por obtener mayores beneficios de los recursos oceánicos se desarrollan los AUVs como herramientas de alto valor en el reconocimiento de lugares inaccesibles. Sus prestaciones brindan una gran aplicabilidad en entornos marinos y su investigación, además de caracterizar el nivel de desarrollo del conocimiento en ramas como la Automática, Informática y la navegación marítima.

Los océanos representan más del 70% de la superficie terrestre y contienen extensas zonas de elevado interés para la humanidad, ya sea con fines científicos, económicos o de preservación medioambiental. Las principales empresas que impulsan el desarrollo de los AUVs están relacionadas con el auge en la explotación de recursos naturales marinos, como el petróleo, el gas natural y las comunicaciones intercontinentales. Por el mercado que generan y las ventajas que ofrecen también se pueden encontrar estos vehículos en escenarios como el desarrollo de investigaciones universitarias, el monitoreo de zonas costeras y el reconocimiento del fondo marino. Los militares por otra parte los sitúan en un nivel estratégico, empleándolos en la detección de minas y misiones de reconocimiento.

En relación a este tema la experiencia en nuestro país se reduce a los trabajos del Grupo de Automatización Robótica y Percepción (GARP) de la Universidad Central de Las Villas (UCLV) en colaboración con el Centro de Investigación y Desarrollo Naval (CIDNAV) con el objetivo de desarrollar un sistema de supervisión y control para un prototipo que lleva por nombre HRC-AUV. Este proyecto está basado en hardware y sensores de bajo costo para que sea viable su producción y explotación. El costo en el mercado internacional de estos productos asciende a cientos de miles de dólares más los gastos en piezas de repuesto y asesoramiento técnico, precios inaccesibles para cualquier empresa nacional. Por esta causa es tan importante el progreso de la actividad colaborativa entre estas dos instituciones, representando una oportunidad para nuestro país de disponer de estos vehículos, además de la adquisición de sólidos conocimiento en el área y el crecimiento del nivel docente en nuestra universidad.

Descripción del entorno de la aplicación

El sistema autopiloto para el HRC-AUV, presenta una arquitectura de la forma computadora-microcontrolador (Guerra Morales, 2010). La computadora empleada está diseñada para uso industrial con arquitectura i586 y compatible con el estándar PC-104; esta se conecta por puerto USB a la Unidad de Mediciones Inerciales (IMU por sus siglas en inglés) mientras que por puerto RS-232 se comunica con el Sistema de Supervisión y con el Sistema de Control. El software de supervisión radica en una laptop industrial en manos del operador. En la PC industrial reside el software de nivel táctico, escrito en lenguaje C sobre un sistema operativo GNU/Linux como una aplicación de usuario, el cual realiza las operaciones de aseguramiento de la autonomía del vehículo, persiste la información de operación para análisis posteriores y funciona como un nodo de comunicación para los demás elementos de cómputo como se puede apreciar en la ilustración No.1.

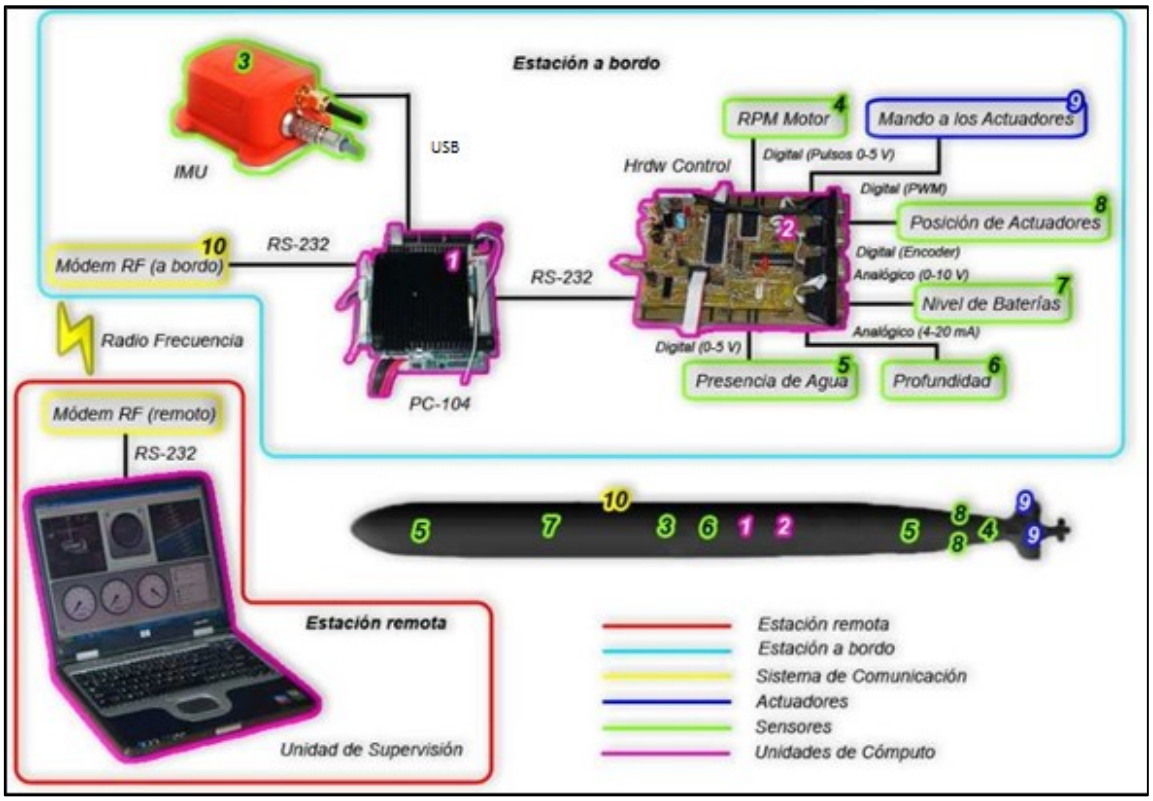

Ilustración 1 Arquitectura de hardware del HRC-AUV

Justificación

El prototipo de software de nivel táctico para el vehículo HRC-AUV de GARP tiene que cumplir con los requisitos de tiempo real dispuestos e incluir todas las funcionalidades necesarias. El GARP tiene que cumplir con el plazo de tiempo convenido para su implementación. Se hace necesaria además la documentación del código para una mejor comprensión y fácil reutilización del mismo o sus componentes en el futuro. Este Trabajo forma parte de investigaciones en desarrollo con vistas a perfeccionar el funcionamiento de este software.

Situación del problema

En los últimos años el GARP ha venido trabajando en el perfeccionamiento del software y sistema de control del HRC-AUV. La versión del software de nivel táctico en el momento de comenzado este trabajo presenta numerosos errores de diseño y problemas funcionales que

en algunos casos elevan la complejidad y la carga de trabajo del procesador. Entre estos se encuentran:

- En tiempo de ejecución se crean demasiados hilos de procesos que dilatan el tiempo de ciclo.
- La interfaz de usuario no está concebida para ejecutarse como servicio y es la única forma de conocer los fallos del sistema.
- Se incumplen los requerimientos de tiempo real en los periodos de ejecución de la tarea de adquisición de datos desde la Unidad de Mediciones Inerciales (IMU) y su transmisión al sistema de control. Como se muestra en la ilustración No. 2, a partir del análisis de datos obtenidos experimentalmente, el periodo se encuentra alrededor de los 13ms, en desacuerdo con las exigencias de diseño de 10ms.

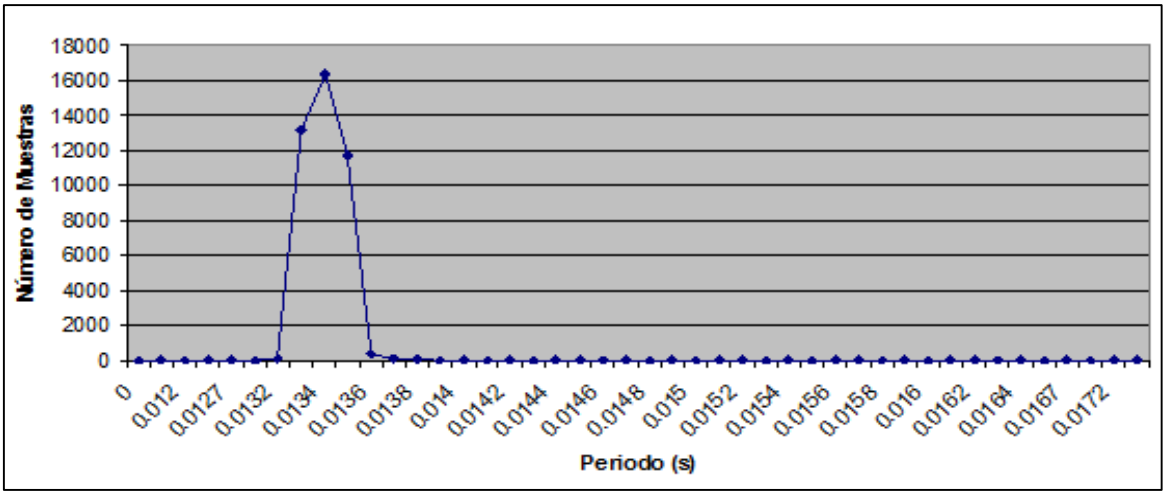

Ilustración 2 Distribución estadística del periodo con que se almacenan los datos históricos para un experimento realizado el 20 de septiembre del 2013

- El código no se encuentra documentado.
- Se emplea una abstracción para elementos de exclusión mutua que evita que se aprovechen los cambios introducidos en el estándar POSIX.1003 de 1996 (bloqueo de lectura-escritura) los cuales beneficiarían el rendimiento de la Base de Datos de Tiempo Real (RTDB) de la aplicación.
- Se realizan operaciones con matrices utilizando una implementación propia.

Los aspectos anteriores impactan en el desempeño del sistema; por lo que, a partir de los problemas mencionados, para el desarrollo de este trabajo se proponen los siguientes objetivos:

Objetivo general

1. Implementar mejoras y nuevas funcionalidades en el prototipo de software de nivel táctico del GARP para AUVs.

Objetivos específicos

1. Identificar en la literatura especializada posibles soluciones a los problemas presentes en el software de nivel táctico del HRC-AUV.

2. Seleccionar métodos y herramientas que permitan la implementación de los cambios necesarios en la aplicación con vistas a:

- Cumplir con los requisitos de tiempo real de la aplicación, en específico el periodo de adquisición de datos desde la IMU.
- Incrementar la calidad del código documentándolo y organizándolo para una mejor reutilización del mismo.
- Seleccionar un nuevo sistema de construcción para la aplicación.

3. Implementar las mejoras y cambios necesarios para dar solución a los problemas presentes en la aplicación.

4. Probar y evaluar el rendimiento de la aplicación mediante el análisis de datos experimentales.

Organización y estructura del trabajo de diploma

El trabajo de diploma se estructura en Introducción, tres capítulos, conclusiones, recomendaciones, referencias bibliográficas y anexos.

CAPÍTULO I

En el primer capítulo se presentan antecedentes teóricos del software empleado en los AUVs. Se realiza una revisión de la literatura especializada para determinar las principales tendencias en la implementación de estas aplicaciones y las técnicas empleadas para lograr características de tiempo real. Se caracteriza la versión en cuestión del programa para identificar posibles mejoras y funcionalidades que tributen al cumplimiento de los objetivos planteados.

CAPÍTULO II

Se seleccionan y describen las herramientas y métodos necesarios para llevar a cabo las modificaciones necesarias, teniendo en cuenta las particularidades y trabajos anteriores del grupo GARP.

CAPÍTULO III

Se describen los resultados del trabajo mediante pruebas experimentales sobre el producto y el análisis de los datos recopilados.

CONCLUSIONES

REFERENCIAS BIBLIOGRÁFICAS

RECOMENDACIONES

ANEXOS

CAPÍTULO 1. ANTECEDENTES TEÓRICOS DEL DESARROLLO DE SOFTWARE PARA AUV

1.1 Introducción

Crear dispositivos que puedan operar de forma autónoma bajo el océano es muy difícil, ya que este es un medio en constante cambio, el cual varía dependiendo de la localización y las condiciones del momento dado. Las características electromagnéticas, densidad, salinidad, temperatura y el movimiento del agua crean un número de obstáculos operacionales que limitan desde el ancho de banda de comunicación hasta la precisión de las mediciones. En la práctica, la mayoría de los grupos que investigan en AUVs tienden a enfocar sus estudios en los paquetes de software y sensores y no en los componentes físicos como la propulsión por ejemplo, que no es fácilmente cambiable (Madden, 2013).

Según (Madden, 2013) existe un creciente rango de publicaciones acerca de los AUVs y sus aplicaciones, que demuestran un mayor interés en el dominio de esta área del conocimiento. También se plantea que en el "*UUV Master Plan*" de La Marina norteamericana (2004), se puede notar una especial atención en el desarrollo de las capacidades de los UUV, así como en la investigación de sus usos potenciales. En el estudio de la literatura especializada se encontraron publicaciones de varias universidades y centros de investigación en áreas de la navegación, comunicación, control, software y componentes, principalmente para reducir los costos tanto como sea posible. Entre estos grupos de investigación se encuentran: "*The Korea Institute of Science and Technology*" (KIOST por sus siglas en inglés), Corea del Sur (Lee et al., 2013); "*Universidad de Oporto*", Portugal (Matos and Cruz, 2009); "*Universidad de Girona*", España (Palomeras et al., 2006); "*Universidad Noruega de Ciencia y Tecnología*" (NTNU) (Fossen, 2002); "*Universidad Nacional de Singapur*", República de Singapur (Chew et al., 2011); "*Wood Hole Oceanographic Institution*", Estados Unidos (Stokey et al.,

2005) y "*Naval Postgraduate School*", Estados Unidos (Roberts, 1997). Además, las competencias entre AUVs incrementan la popularidad de los mismos e incentiva el progreso y perfeccionamiento de estos, como ejemplo se incluyen la "*Student Autonomous Underwater Vehicle Challenge-Europe"* (SAUC-E) y la "*AUVSI Robosub Competition*" en Australia. Desde una perspectiva nacional, el grupo GARP ostenta varias publicaciones y estudios de gran relevancia entorno al trabajo en el diseño y perfeccionamiento del sistema de supervisión y control del HRC-AUV. Entre estas se encuentran trabajos sobre modelado y control del vehículo (Cañizares, 2010), diseño e implementación del sistema de control (Guerra Morales, 2010), arquitectura de software y hardware (Martinez et al., 2010), software de navegación y guiado (Lemus Ramos, 2011) y mejoramiento de las prestaciones de la IMU (García García, 2010).

1.2 Arquitectura de software para robots

En el presente epígrafe se refieren las principales arquitecturas de software de forma general y organizativa según su evolución histórica. Se hace énfasis en la estructura que presentan y las funciones de cada nivel de abstracción.

La primera arquitectura de software de control para robots autónomos quizás fue *"Sense-Plan-Act"* (SPA), la cual definía un sistema de control para robots autónomos móviles en la descomposición de elementos funcionales: un sistema sensor, uno de planeación y otro de ejecución. La función del sistema sensor es transformar las salidas de los sensores a un modelo. El planificador emplea como entrada este modelo y un objetivo definido, entonces genera un plan para arribar a este último. El sistema de ejecución toma este plan y realiza las operaciones necesarias en el robot para alcanzar el objetivo del plan (Foster et al., 2006).

En la década de 1990 la arquitectura *"Subsumption"* fue una alternativa a las anteriores. Esta es un protocolo computacional paralelo y distribuido para conectar sensores a actuadores en robots. Una arquitectura *"Subsumption"* es una forma de descomponer comportamientos inteligentes complicados en varios módulos más simples, los cuales están organizados en capas. Cada capa implementa un objetivo particular del agente y las capas superiores son cada vez más abstractas (Foster et al., 2006).

Cada módulo de esta arquitectura tiene implícito el modelo del medio y es invocado directamente por sensores. Una dificultad con este método surge cuando dos o más comportamientos o formas de actuar están activos y estos compiten por los recursos del sistema, que en la versión pura de *"Subsumption"* siempre se favorece a la necesidad más urgente y la menos importante queda en segundo plano.

Mientras esta arquitectura depende de la comprensión del mapeo implícito de bajo nivel de los comportamientos para alcanzar los objetivos de alto nivel, el modelo que a continuación se refiere: *Rational Behavioral Model* está basado en una estrategia de control de alto nivel que se plantea evadir estos conflictos entre comportamientos haciendo uso de un conjunto inherente de reglas impuestas por el objetivo principal (Foster et al., 2006).

Rational Behavioral Model

El "Modelo de Comportamiento Racional" (RBM por sus siglas en inglés) es un modelo para software de control de robots móviles que presenta una jerarquía con tres niveles, el estratégico, el táctico y el nivel de ejecución como se aprecia en la ilustración No.3. Estos últimos pueden ser implementados en diferentes lenguajes de programación para facilitar el desarrollo y cumplimiento del esquema del modelo (Kwak et al., 1992).

El nivel estratégico está en la cima de la jerarquía y mantiene las doctrinas operacionales del vehículo. Estas doctrinas describen en alto nivel, es decir, términos entendibles por humanos, el comportamiento "racional" permisible del vehículo.

El nivel de ejecución es el último de la jerarquía. Es responsable de la interacción con los sensores y actuadores del vehículo. Este puede incluir ejecución de lazos de servo control y limitada integración de datos de sensores.

El nivel táctico o intermedio tiene varias tareas. Asimila datos de componentes del nivel de ejecución en alto nivel como una descripción adecuada del "estado del vehículo" para ser utilizada en el nivel estratégico. Toma directivas generales (comandos de comportamiento) del nivel estratégico y elabora comandos específicos para el nivel de ejecución.

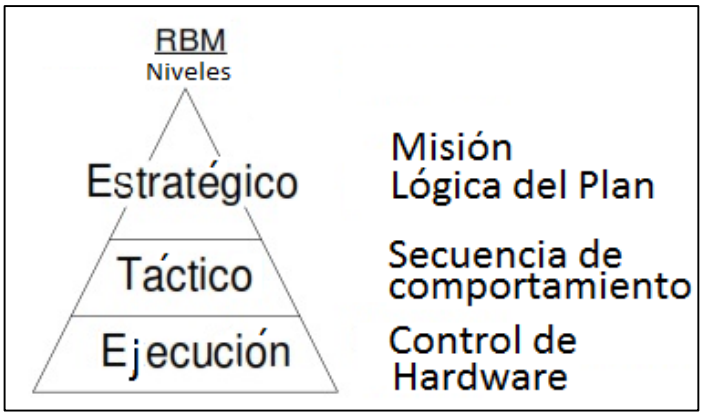

Ilustración 3 Arquitectura de RBM

Arquitectura de software distribuida (DVMA)

La arquitectura de software distribuida (DVMA por sus siglas en inglés) consiste en tres niveles jerárquicos de módulos de software: el de hardware, el funcional y el de comportamiento. El nivel de hardware es el más bajo, es un grupo de módulos de software que controlan directamente las interfaces con el hardware. El nivel intermedio o funcional consiste en módulos de software para el proceso de organizar comportamientos específicos que son llamados módulos funcionales. Un proceso que comunica con el nivel de comportamiento, llamado módulo de comunicación inter-capa es también incluido en este nivel. La capa más alta, es donde se determina el estado del comportamiento del vehículo, llamado modo de comportamiento. Los módulos en esta capa toman decisiones según la situación, comunicando y enviando comandos al correspondiente módulo funcional para extraer procesos necesarios y determinar el modo de comportamiento. Debe notarse que el cambio de comportamiento solo ocurre en el nivel de comportamiento independientemente de otros niveles. Esto conlleva a un mecanismo simplificado para misiones complejas (Fujii et al., 1993).

1.3 Ejemplos de AUVs

Phoenix AUV

Es un robot diseñado en el *Naval Postgraduate School* para investigaciones docentes en Control y sensores en aguas superficiales. Presenta 3 niveles en la arquitectura de software, ejecución, táctico y estratégico según el paradigma *Rational Behavior Model* (RBM). El nivel de ejecución se encarga del control de los actuadores, corre en un GesPac 68030 sobre OS-9, está escrito en (K&R) C y diseñado para ejecutarse en tiempo real. Los niveles táctico y estratégico residen en una laptop Voyager Sparc 5 sobre Solaris, escritos en ANSI C y Prolog respectivamente. El primero de ellos es el responsable de la interacción con los sensores y el procesamiento de la información y está diseñado para ejecutarse en tiempo real suave y el segundo realiza el planeamiento de la misión en lapsos de tiempo mayores (Brutzman et al., 1996).

ISiMI6000

El *ISiMI6000* es un vehículo para la observación del fondo oceánico desarrollado en Corea del Sur. Cuenta con dos computadores industriales compatibles con el estándar PC-104 Intel Atom N450 1.66 GHz para control y navegación y dos microcontroladores para controlar las interfaces analógica-digital y digital-analógica. Dispone además de un sistema operativo que no es para tiempo real: Windows 7. La arquitectura de control ha sido implementada en un archivo ejecutable en C, pero corre como modo agente, modo control y modo navegación. El primero sobre un ordenador agente y los dos últimos sobre cada PC industrial. La aplicación se sincroniza con temporizadores definidos por el usuario por software, implementados en hilos en el trasfondo y aplicando el sistema de tiempo de la Computadora de Tarjeta Simple (SBC por sus siglas en inglés) sin presentar fallas (Lee et al., 2013).

En la PC de control, el proceso principal tiene 4 pasos: actualizar la entrada de datos, Pequeño Lenguaje de Misión (TML por sus siglas en inglés), ejecución y espera hasta que se cumplan los 10ms de periodo.

Según (Lee et al., 2013) para verificar el proceso de tiempo real se realizaron experimentos en Windows 7 y los resultados mostraron que el promedio de error en los periodos de tiempo fue casi 0 cuando la frecuencia deseada de control estaba entre 10Hz y 100Hz. Utilizar el temporizador por software y no un RTOS es suficiente para controlar el ISiMI6000 a la frecuencia deseada. El temporizador no es capaz de operar en tiempo real duro, pero si en tiempo real suave. Esto es suficiente para la mayoría de las aplicaciones.

Starfish

El AUV *Starfish* utiliza una arquitectura de software distribuida para vehículos autónomos (DSAAV por sus siglas en inglés) que provee una flexible reconfiguración del software. DSAAV emplea atención a llamadas remotas que permiten una implementación distribuida de los componentes del software dentro del AUV. Esto hace posible con mayor facilidad una reimplementación e incluso una migración de los componentes del software hacia otra plataforma, por ejemplo de PC industriales con estándar PC-104 a Microcontrolador (MCU por sus siglas en inglés) (Chew et al., 2011).

Mares

El AUV *Mares* está basado en un sistema computacional con una tarjeta madre principal (ilustraciones No. 4 y No. 5) y otras dos para la interfaz con periféricos, como monitores del sistema, dispositivos actuadores y navegación y sensores. El software fue desarrollado en C++ sobre un kernel de Linux y está compuesto por un conjunto de procesos independientes. La comunicación entre los módulos se basa en un mecanismo de pase de mensajes empleando *User Datagram Protocol* (UDP). La interfaz con el hardware es administrada por procesos dedicados que proveen una capa de abstracción, que obtiene los datos de los sensores de navegación y el sistema transmite estos datos al módulo de navegación (ilustración No. 6). Este implementa todos los algoritmos necesarios para estimar el estado del AUV en tiempo real, también envía la estimación al módulo de control. El módulo de control es responsable de la ejecución de la misión. En cada ciclo de control se verifica la terminación de la maniobra

actual y planifica la siguiente. Se registran los mensajes del sistema donde se encuentra toda la información relacionada con la operación, los sensores y el sistema (Matos and Cruz, 2009).

Ilustración 4 Unidades de cómputo del MARES

Ilustración 5 AUV MARES

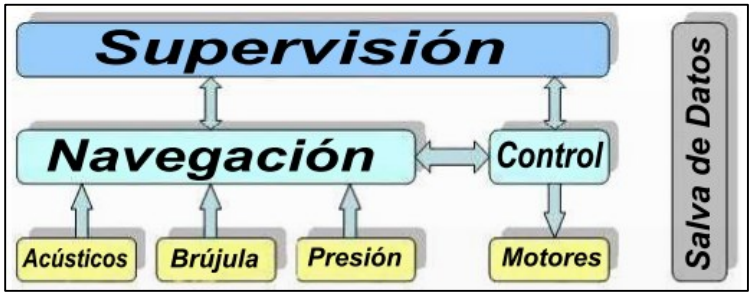

Ilustración 6 Estructura general del MARES

Twin-Burger

Diseñado en el Instituto de Ciencia Industrial de la universidad de Tokyo como base de prueba y verificación de software en ambientes reales. En este vehículo se propone una arquitectura de software distribuida (DVMA) que genera una secuencia de comportamientos para comandar la misión mientras se maneja la interfaz de hardware en tiempo real. Arquitecturas de multiprocesador y procesamiento en paralelo son satisfactorias para mantener el proceso en tiempo real mientras se ejecuta software de alto nivel. Se utiliza un sistema multiprocesador basado en INMOS (Fujii et al., 1993).

Yellowfin

Desarrollado en el *Georgia Tech Research Institute*, el procesamiento está dividido en dos secciones: un procesador de bajo nivel, que opera todo el hardware y software de los sensores y actuadores y un procesador de alto nivel que opera el comportamiento autónomo y colaborativo. Tiene un paquete de software completo desde planificación pre-misión hasta ejecución de misión. La planificación de pre-misión se ejecuta por *Mission Lab,* por comportamientos disponibles para generar comportamientos de misión. El nivel de ejecución se desarrolla utilizando MOOS-IvP como se muestra en la ilustración No. 7, un grupo de aplicaciones para alto nivel y un controlador de bajo nivel XMOS. Las órdenes y control son realizadas en la estación base utilizando *Falcon View*. Una misión puede ser ejecutada en el simulador o en el vehículo real (West et al., 2010).

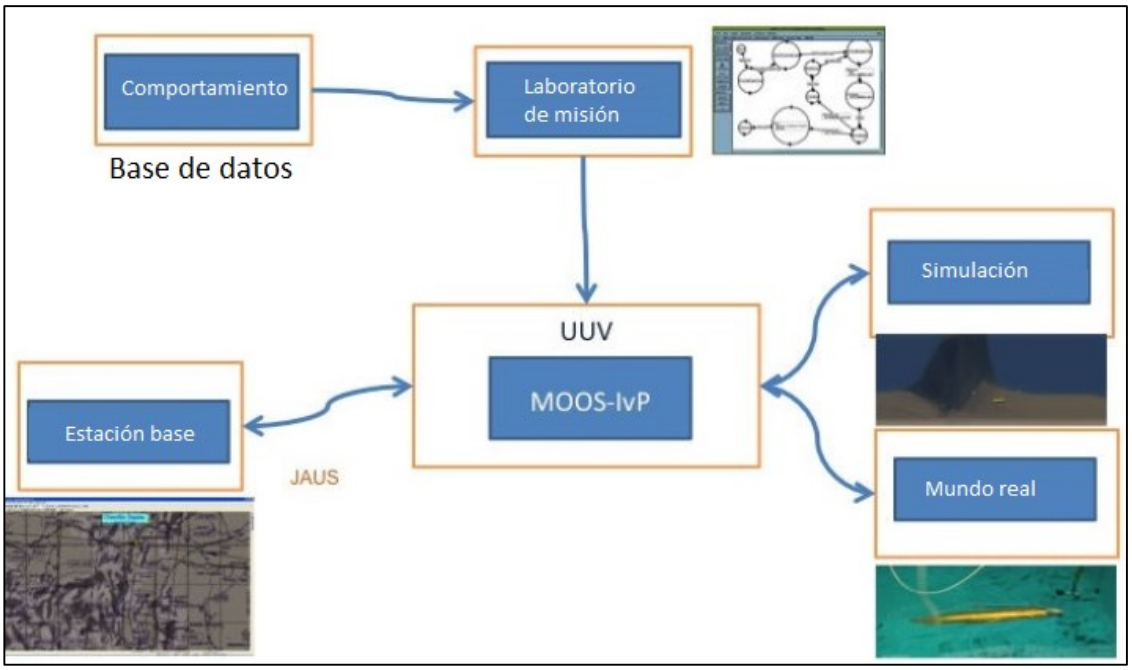

Ilustración 7 Esquema de interacción entre módulos en el Yellowfin

1.4 Sistema Operativo

La selección de un Sistema Operativo (OS por sus siglas en inglés) para su uso en AUVs es a menudo basado en las habilidades, experiencias y software disponible de proyectos anteriores. La idea es reutilizar tantas herramientas como sea posible, por eso muchos grupos de proyecto aprovechan softwares libres como Ubuntu; o QNX, si los diseñadores deciden que se requiere tiempo real (Madden, 2013).

Utilizar un SO de código fuente abierto es considerado por muchos grupos de investigadores porque estos proveen una flexibilidad elevada de los parámetros de bajo nivel del Sistema Operativo, facilita la reutilización de código y lo más importante, la licencia para su uso no requiere de pago.

1.4.1 MetaOS

Muchos Sistemas Operativos de tiempo real son ampliamente aprovechados para tareas genéricas, pero pueden ser usados con paquetes de software específicos que proveen típicamente funciones como generación de movimiento robótico o procesamiento de sensores. Existe un término para llamar a este software: "metaOS", ya que a menudo proveen niveles de abstracción para acceder a drivers de dispositivos de bajo nivel como sensores y actuadores y son desarrollados por comunidades interesadas en compartir recursos (software libre). Estos se diferencian de las librerías de código en que las librerías solo proporcionan un conjunto de estructuras de datos y funciones que pueden ser usadas según se necesiten, no especifican como está estructurado el programa o como funciona. El metaOS permite a diseñadores de vehículos enfocarse en el desarrollo de nuevos o únicos componentes en sus sistemas, que pueden ser incorporados en futuras versiones del metaOS. Algunos de los más comunes son destinados específicamente a la robótica (Madden, 2013).

Un metaOS (meta Sistema Operativo) es un paquete de software con funcionalidades para coordinar la utilización de componentes importantes del sistema como sensores o dispositivos de comunicación y procesamiento de la información disponible (Madden, 2013).

Robot Operating System (ROS)

Es un metaOS de código abierto diseñado específicamente para apoyar la reutilización de código en aplicaciones para robótica. Está soportado actualmente por Willow Garage y varias organizaciones en el campo de la robótica. ROS funciona como un conjunto de procesos punto a punto o nodos, que son conectados usando la infraestructura de comunicación ROS, la cual especifica el formato de comunicación usado para enviar información entre los nodos. El nodo provee el control de bajo nivel de dispositivos y capas de abstracción de hardware, con el paso de mensajes y paquetes de administración de manera similar a muchos otros frameworks en robótica. Este también facilita herramientas y librerías para el desarrollo de código a través de múltiples procesos. ROS está dedicado fundamentalmente para Ubuntu y Mac OS X, aunque muchos otros basados en Linux están soportados por la comunidad. El código ha sido fundamentalmente desarrollado en Paython y C++, aunque otras librerías experimentales están siendo desarrolladas. ROS puede también integrarse con otros metaOS y frameworks (Kraetzschmar et al., 2010).

Mission Oriented Operating Suite (MOOS)

Fue inicialmente desarrollado por Paul Newman como un metaOS para aplicaciones en robótica, destinado principalmente a AUVs. El mismo ha crecido en una comunidad soportando sistemas que proveen un lazo de control basado en estructuras multiplataforma. Este comienza basado en una topología que utiliza una base de datos central llamada MOOSDB, la cual conecta a cada aplicación para facilitar el mensaje apropiado en una cadena leíble por humanos o en formato de doble precisión. Más recientemente la arquitectura de control Helm ha sido añadida para crear MOOS-IvP. Esta combinación construida en MOOS facilita el control automatizado de robots basado en comportamiento. Una de las limitaciones de MOOS es que su inherente estructura basada en lazo no permite garantizar un comportamiento para tiempo real o un correcto funcionamiento con los datos de sensores obtenidos con diferente frecuencia (Madden, 2013).

Universal Robot Body Interface (URBI)

Ha sido desarrollado por Gostai y está disponible como un metaOS de código abierto multiplataforma destinado al desarrollo de sistemas complejos principalmente en la robótica. URBI está basado en UObject, una arquitectura distribuida integrada por C++, la cual utiliza urbiscript, un lenguaje interpretado y orientado a eventos que permite ser desarrollado como un sistema multiplataforma y es empotrado en varias tarjetas de CPU. URBI es interoperable con ROS y otros softwares basados en C++ (Madden, 2013).

Player

Es un metaOS de código abierto muy empleado en la comunidad robótica que proporciona un estilo de interfaz de red para conectar un rango de robots y hardware de sensores. Aplica un modelo cliente/servidor, que permite a los desarrolladores de software emplear varios lenguajes de programación y correr sobre cualquier computadora que tenga una conexión de red a la señal de mensaje de los componentes del robot. Aunque Player fue inicialmente desarrollado entorno a la plataforma robótica Pioneer 2, soporta ahora un elevado rango de hardware robótico (Kraetzschmar et al., 2010).

1.5 Sistemas de Tiempo Real

Un Sistema de Tiempo Real es aquel cuyo desempeño no depende solo de los resultados de sus cálculos y procedimientos computacionales, sino que además depende del instante de tiempo en el que estos se producen (Kopetz, 2011).

Para su estudio, se descompone en tres subsistemas:

- **Subsistema Controlado**: está conformado por el objeto de control (planta, sensores, actuadores).
- **Subsistema Computacional**: Está conformado por el sistema computacional de tiempo real.
- **Subsistema Operador**: Está conformado por el operador humano.

1.5.1 Clasificaciones de los Sistemas de Tiempo Real

De acuerdo a sus características, los Sistemas de Tiempo Real se pueden clasificar en: Tiempo Real Fuerte (HRT por sus siglas en inglés) y Tiempo Real Suave (SRT por sus siglas en inglés) (Kopetz, 2011).

El diseño de sistemas de Tiempo Real Fuerte es diferente de los sistemas de Tiempo Real Suave por los requisitos y tolerancias inherentes a ambos.

Tiempo de Respuesta: Las demandas de tiempo de respuesta en los sistemas HRT, son generalmente del orden de los milisegundos o menos, debe ser respetada en todo momento y requiere autonomía de la intervención de operadores tanto en operación normal como en situaciones críticas. Los sistemas con requerimientos de SRT, por el contrario, presentan tiempos de respuesta en el orden de los segundos, y si se incumple un plazo de entrega, los resultados no son graves.

Desempeño ante picos de carga: En los sistemas HRT, el escenario de los picos de carga debe estar bien definido, se debe garantizar que la arquitectura computacional del sistema permita el cumplimiento de los plazos de entrega a pesar de los picos. En los sistemas SRT, el desempeño medio del sistema es lo realmente importante, y la degradación de las condiciones de operación a razón de un pico de carga que ocurre de forma aislada, es tolerada por razones económicas.

Sincronismo: El sistema de HRT, debe mantenerse sincronizado con su medio ambiente (objeto de control y operador), ante todos los posibles escenarios. Los SRT, en cambio, pueden ejercer algún tipo de control sobre el medio (por ejemplo, modificar la velocidad de respuesta) en caso que no se pueda procesar la carga de datos requerida.

Determinismo Temporal: Es fundamental que el comportamiento temporal de los sistemas de tiempo real sea determinista. Esto no significa que sea eficiente, sino que el sistema debe responder correctamente en todas las situaciones, previendo el comportamiento en el peor caso posible.

Fiabilidad y seguridad: Un fallo en un sistema de control puede hacer que el sistema controlado se comporte de forma peligrosa o antieconómica. Es importante asegurar que si

el sistema de control falla lo haga de forma que el sistema controlado quede en un estado seguro, es decir, teniendo en cuenta los posibles fallos o excepciones en el diseño.

Concurrencia: Los componentes del sistema controlado funcionan simultáneamente. El sistema de control debe atenderlo y generar las acciones de control simultáneamente. En este caso existen dos opciones: sistemas monoprocesador con procesos multihilos o computadoras multiprocesador.

1.5.2 Programación

En la programación de sistemas de tiempo real son varias las dificultades técnicas afrontadas al emplear librerías de bajo nivel del lenguaje C, incluso en la implementación de una simple tarea cíclica (Buttazzo and Lipari, 2013).

Existen implementaciones de librerías de código abierto en C que simplifican la programación de tiempo real en sistemas Linux, ocultando detalles de bajo nivel de creación de tareas, localización y sincronización y ofrecen utilidades para funcionalidades de más alto nivel (Buttazzo and Lipari, 2013).

Hilos

Los hilos o procesos "ligeros", son subprocesos de control independientes dentro de un proceso, que comparten datos globales pero mantienen su propia pila, variables globales y contadores de programa. Los hilos son relacionados con procesos "ligeros" porque su contexto es menor que el de los procesos, que tienen espacios de memoria totalmente independiente. Por lo tanto, el cambio de contexto entre hilos puede ser más económico. Además, los hilos son un modelo adecuado, simple pero poderoso, que permite sacar provecho del paralelismo en un entorno multiproceso con memoria compartida, es decir, la creación de un nuevo hilo es una característica que permite a una aplicación realizar varias tareas a la vez (concurrencia) (Mueller, 1993).

API

Típicamente las aplicaciones son programadas contra una Interfaz de Programación de Aplicaciones (API por sus siglas en inglés), no directamente a llamadas al sistema. Ello resulta importante debido a que no se necesita una correlación directa entre las interfaces utilizadas por las aplicaciones y la interfaz real brindada por el kernel. Una API define un grupo de interfaces usadas por aplicaciones. Las mismas pueden ser implementadas como una llamada al sistema, a través de múltiples llamadas al sistema o sin el uso de llamadas al sistema. Una misma API puede existir en múltiples sistemas y brindar la misma interfaz a aplicaciones, mientras que la implementación de la misma API puede cambiar de sistema a sistema (Mueller, 1993).

1.6 Conclusiones parciales del capítulo

El desarrollo a nivel mundial en tema de software para Vehículos Autónomos Sumergibles y el perfeccionamiento y creación de bibliotecas que optimizan y facilitan la programación de sistemas de tiempo real brindan las herramientas necesarias para la implementación de mejoras en el software de nivel táctico para AUV de GARP, aprovechando el hecho de que muchos de estos proyectos son de software libre y se encuentran ampliamente difundidos.

Teniendo en cuenta que la tendencia en la implementación de este tipo de proyectos a nivel mundial se ha generalizado en el empleo de meta Sistemas Operativos de código fuente abierto con amplias librerías especializadas en robótica móvil y particularmente en AUV que facilitan el trabajo del programador, se concluye que la línea de trabajo que rige el desarrollo de software en el grupo GARP no se encuentra actualizada en relación con la adoptada en la bibliografía consultada.

Aunque el Sistema Operativo instalado en el HRC-AUV no es especializado en tareas de tiempo real, sí se encontraron referencias que indican que este puede cumplir con los periodos de tiempo necesarios.

CAPÍTULO 2. HERRAMIENTAS Y TÉCNICAS DE IMPLEMENTACIÓN

2.1 Introducción

En este capítulo se presentan las herramientas de software y las técnicas empleadas para la implementación de la aplicación, así como sus principales características. Se abordan además las soluciones ofrecidas para los problemas planteados, así como el procedimiento que se aplicó en cada caso.

2.2 Entorno de ejecución

El entorno de ejecución comprende el hardware y el software sobre el que se desarrolla la aplicación, provee servicios para un programa en ejecución. De forma general este define cuáles facilidades (librerías, funcionalidades y otros programas y servicios) están disponibles al desarrollador de la aplicación por adelantado, permitiéndole explotar todo el potencial que le ofrece.

En este trabajo se aborda solo lo referente al software, ya que no es objetivo del mismo el análisis del hardware del vehículo HRC-AUV, por dos razones: se encuentra descrito en publicaciones del grupo y porque se ha seleccionado de forma tal que cumple con un estándar industrial el cual se encuentra ampliamente documentado

Requisitos del entorno de ejecución

- Debe permitir analizar el comportamiento temporal del producto.
- Debe garantizar el cumplimiento de los requisitos temporales.

- Debe garantizar que no existan restricciones de memoria u otras que afecten a la velocidad.
- Debe ser lo más sencillo posible.

Los perfiles de ejecución pueden estar orientados especialmente para los sistemas de tiempo real como el perfil de Ravenscar, que es un subconjunto del lenguaje de programación Ada que impone ciertas restricciones a la parte concurrente del lenguaje para poder realizar análisis temporales y permitir una implementación eficiente del núcleo de ejecución. Teniendo en cuenta la selección de la arquitectura de hardware y Sistema Operativo realizado previamente (Lemus Ramos, 2011), la mejor opción es tomar como base las normas POSIX, específicamente POSIX.1003-1996 (Joint Technical Committee ISO/IEC JTC 1, 1999), la cual pretende ofrecer una descripción unificada de todos los sistemas compatibles con UNIX que permitan la portabilidad de aplicaciones entre ellos y además es un superconjunto del estándar C.

Con esta decisión, el equipo de desarrolladores tiene la posibilidad de seleccionar con posterioridad otro Sistema Operativo en caso de verificarse que el seleccionado no cumple con los requisitos de tiempo real impuestos. Teniendo en cuenta el estudio realizado en (Lemus Ramos, 2011), los sistemas GNU/Linux y por tanto el Sistema Operativo Debian, seleccionado para este prototipo, no son la base más adecuada para aplicaciones que tienen restricciones de tiempo real (ni débiles ni fuertes). La posibilidad de portar la aplicación, en caso de ser necesario, a otros sistemas operativos más aptos en situaciones críticas de tiempo real, como son QNX y VxWorks (ambos cumplen con la norma antes mencionada), representa una ventaja estratégica. Es válido aclarar que GNU/Linux cumple con la norma desde el punto de vista técnico, pero no legalmente, pues esto requeriría de un desembolso de dinero que la comunidad de desarrolladores y su equipo legal considera innecesario.

POSIX

Interfaz de Sistema Operativo Portable (POSIX por sus siglas en inglés, y la X viene de UNIX como señala la identidad de la API).

El estándar POSIX es una familia de normas que persiguen generalizar las interfaces de los sistemas operativos para que una forma de aplicación pueda ejecutarse en distintas plataformas. Surgieron de un proyecto de normalización de las API y describen un conjunto de interfaces de aplicación adaptables a una gran variedad de sistemas operativos. El objetivo consiste en reducir la cantidad de esfuerzo a la hora de portar una aplicación con facilidad a sistemas como UNIX, LynuxOS y MINIX. Define perfiles de aplicación como:

POSIX 13: Perfiles para sistemas de tiempo real.

- PSE50: Sistema de tiempo real mínimo.
- PSE51: Controlador de tiempo real.
- PSE52: Sistema de tiempo real dedicado.
- PSE53: Sistema de tiempo real generalizado.

La extensión POSIX para hilos especifica un modelo de hilo dirigido por prioridades con políticas de planificación reentrante, manejo de señales y servicios de exclusión mutua, así como espera sincronizada (Mueller, 1993).

El estándar Pthreads brinda una base uniforme para aplicaciones multiprocesos con memoria compartida, sistemas de tiempo real y un modelo económico para programas multihilo sobre un solo procesador. El concepto de hilo puede ser empleado para implementar tareas en Ada o para expresar paralelismo dentro de una aplicación en el nivel de lenguaje de programación (Barney, 2009). Una implementación de Pthreads puede ser llevada a cabo como:

- Implementación de kernel, donde funcionalmente todo es parte del kernel del sistema operativo.
- Implementación de librería, donde funcionalmente todo es parte de un programa usuario y pueden enlazarse internamente.
- Una mezcla de las anteriores.

2.3 Librerías para cálculo científico

El uso de implementaciones propias en las operaciones con matrices no se considera una buena práctica de programación teniendo en cuenta que existen bibliotecas estándares de código abierto que contienen las funciones necesarias para la solución de problemas de álgebra lineal.

Subrutinas Básicas de Álgebra Lineal (BLAS)

Las Subrutinas Básicas de Álgebra Lineal (BLAS por sus siglas en inglés) son una serie de rutinas organizadas en tres niveles con el objetivo de proveer implementaciones eficientes, mantenibles, modulares y portables de algoritmos para computadoras de alto rendimiento; especialmente para aquellas que posean memoria jerárquica (RAM, ROM, caché) y capacidades de procesamiento paralelo (Dongarra et al., 1990).

BLAS fue publicado por primera vez como una librería de Fortran en 1979 debido a la necesidad de adoptar un grupo de rutinas básicas para resolver problemas de álgebra lineal; sin embargo, una interfaz estándar para C ha sido desarrollada, denominada CBLAS (Lawson, 1999). BLAS se utiliza en lenguajes de programación de alto y bajo nivel en implementaciones de librerías y programación algebraica. Actualmente constituyen la Interfaz de Programación de Aplicaciones (API por sus siglas en inglés) estándar para rutinas y librerías de álgebra lineal. Varias implementaciones de BLAS han sido adaptadas para arquitecturas de computadoras específicas, por ejemplo las desarrolladas para las compañías de Intel y AMD (Datardina et al., 1992).

Librería Científica de GNU (GSL)

La Librería Científica de GNU (GSL por sus siglas en inglés) es una colección de rutinas para el cálculo numérico. Las rutinas se han escrito desde cero en C, y presentan una moderna API para programadores de C, permitiendo que varias capas puedan ser escritas por lenguajes de muy alto nivel (Galassi et al., 2009). El código fuente es distribuido bajo la Licencia Pública General de GNU (GNU GPL por sus siglas en inglés). La librería cubre un gran rango

de tópicos en la computación numérica. Entre las rutinas disponibles se encuentran las siguientes áreas:

BLAS	Transformada rápida de Fourier
Números complejos	Integración numérica
Vectores y matrices	Derivación numérica
Polinomios	Funciones especiales
Ecuaciones diferenciales ordinarias	Aritmética de punto flotante IEEE

En este proyecto se decide emplear la Librería Científica de GNU por ofrecer una interfaz más amigable y ser la librería estándar para el trabajo algebraico.

2.4 Persistencia de Información de Depurado

Para dar solución al problema de persistencia de información de depurado se decidió emplear la facilidad syslog. Esta es un estándar para el envío de mensajes de registro en una red informática basada en IP. Además se conoce como la aplicación o biblioteca que envía los mensajes de registro. Es el formato en el que se guarda la información del evento en un fichero que generalmente en los sistemas UNIX es almacenado en el directorio /var/log y puede ser modificado si se desea. Existe un archivo mayor que guarda los registros, por lo que se pueden monitorear los mensajes registrados en /var/log/message. Al igual que syslogd o rsyslogd generalmente se ejecuta como demonio y es ampliamente utilizado en la obtención de información de depurado en aplicaciones que se ejecutan como servicios.

Cuando invocamos la función syslog() deseamos registrar un evento que puede ocurrir durante la ejecución de una aplicación. Para esto se requieren dos parámetros, el nivel de importancia o seriedad (tabla 2.1) y una cadena de caracteres. El argumento de prioridad es una combinación del nivel de importancia y la facilidad, este último, argumento de la función openlog().

Nivel de seriedad	**Descripción**
LOG_EMERG	Emergencia (sistema inutilizable)
LOG_ALERT	Condición que debe ser arreglada inmediatamente
LOG_CRIT	Condición critica
LOG_ERR	Condición de error
LOG_WARNING	Condición de advertencia
LOG_NOTICE	Normal, pero condición significativa
LOG_INFO	Informe de mensaje
LOG_DEBUG	Mensaje de depurado

Tabla 2.1 Niveles de syslog ordenados de mayor a menor

Se reemplazaron todas las llamadas a funciones de envío de información a la salida estándar y todas aquellas funciones que enmascaraban llamadas a las funciones de envío a la salida estándar por llamadas a la función (syslog.h: syslog()). En la Tabla 2.2 se detallan todos los cambios realizados indicando el módulo de código modificado y una descripción de los cambios realizados:

Módulo	**Comentario**
src/dspic_comunication.c	Reemplazadas 6 llamadas a la función (src/display.c: LogMessageToScreen()) por llamadas a la función (syslog.h: syslog()).

src/main.c	Reemplazadas 2 llamadas a la función (src/display.c: LogMessageToScreen()) por llamadas a la función (syslog.h: syslog()). Reemplazadas 2 llamadas a la función (stdio.h: printf()) por llamadas a la función (syslog.h: syslog()).
src/mti_comunication.c	Reemplazada 1 llamada a la función (src/display.c: LogMessageToScreen()) por llamada a la función (syslog.h: syslog()).
src/mti_comunication.c	Reemplazadas 4 llamadas a la función (src/display.c: LogMessageToScreen()) por llamadas a la función (syslog.h: syslog()).
src/remote_comunication.c	Reemplazadas 7 llamadas a la función (src/display.c: LogMessageToScreen()) por llamadas a la función (syslog.h: syslog()).
src/matrix.c	Reemplazadas 5 llamadas a la función (stdio.h: printf()) por llamadas a la función (syslog.h: syslog()).
src/system.c	Reemplazadas 4 llamadas a la función (stdio.h: printf()) por llamadas a la función (syslog.h: syslog()).

Tabla 2.2 Cambios para solucionar el problema de persistencia de información de depurado.

2.5 Comportamiento de Demonio

Un demonio es una clase especial de proceso en sistemas compatibles con el estándar POSIX.1003 el cual no tiene una Terminal de control; por tanto, se dice que se ejecuta en el trasfondo (Rago and Stevens, 2005).

El software de nivel táctico del HRC-AUV debe ejecutarse durante todo el tiempo que el vehículo se encuentra en funcionamiento, sin depender para ello de intervención de actor

alguno (tanto humano como autómata), para lo cual debe comportarse como un servicio. Con este objetivo se introdujo en la aplicación un nuevo módulo de código (src/daemon.c) el cual implementa dos nuevas funciones públicas y se muestra en el ANEXO C.

Una de ellas (src/daemon.c: void daemonize(const char *cmd)) hace que la aplicación se comporte adecuadamente como un demonio. La otra (src/daemon.c: int already_running(const char * lock_filename)) garantiza que solo una copia de la aplicación de navegación se pueda ejecutar en cada instante de tiempo, asegurando de esta forma la exclusión mutua de los recursos ofrecidos por el Sistema Operativo a través del Sistema de Archivos. La última funcionalidad previene los inconvenientes causados en el caso de que el desarrollador introduzca un cambio en el código de la aplicación y lo aplique sin percatarse de que ya existe una copia de la aplicación en ejecución. En este caso ninguna de las dos copias de la aplicación recibiría paquetes de datos desde la IMU o desde el Sistema Supervisor, volviendo no operativas ambas copias.

2.6 Sistema de construcción

Con el fin de alcanzar un alto grado de portabilidad a nivel de código fuente se selecciona como sistema de construcción el “Sistema de Construcción GNU” (GNU Build System en inglés), también conocido como “Autotool”. Es un conjunto de herramientas que ayudan a la portabilidad de aplicaciones que se encuentran en uso en la mayoría de los proyectos de código abierto del mundo. Tiene dos objetivos, simplificar el desarrollo de programas portables y simplificar la construcción de programas que son distribuidos como código fuente. Para construir un programa desde código fuente, el instalador solo necesita la utilidad make, un compilador, un Shell y en algunos casos utilidades estándares de UNIX como: sed, awk, yacc, lex.

Cuando hablamos de Autotool nos referimos principalmente a los tres siguientes paquetes de herramientas:

1. **Autoconf:** Produce un fichero Shell de configuración llamado *configure* el cual es el encargado de realizar las comprobaciones para determinar la disponibilidad y ubicación de todos los requisitos en la construcción exitosa de un programa; información relacionada a la portabilidad que es requerida para adaptar los archivos *Makefile* de configuración de cabeceras y otros archivos específicos. Entonces este

procede a generar una versión personalizada de estos archivos a partir de plantillas genéricas. De esta forma el usuario no tiene que personalizar estos archivos manualmente. El proceso real de *configure* es mostrado en la ilustración No. 8.

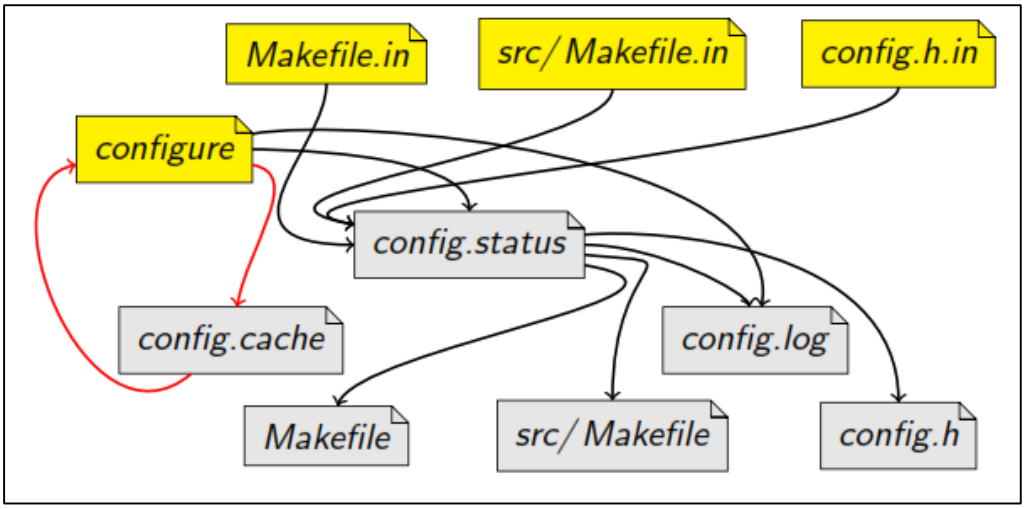

Ilustración 8 Proceso real de *configure*

2. **Automake:** Es una herramienta que permite generar en forma automática archivos *Makefile*, de acuerdo al estándar de GNU Makefile, simplificando el proceso de organización de las distintas reglas así como suministrando funciones adicionales que permiten un mejor control sobre los programas (Calcote, 2010).
3. **Libtool:** Es una interfaz de línea de comandos para el compilador y enlazador que hace más sencilla la generación de librerías estáticas y compartidas de forma portable. Típicamente se utiliza con Autoconf y Automake, pero puede ser aplicada independientemente. Simplifica el trabajo de los desarrolladores mediante el encapsulado de las dependencias específicas de la plataforma, y la interfaz del usuario en un único script (Calcote, 2010).

Existen otras herramientas como Aclocal, Autoheader y Autotoolset que en conjunto con las anteriores reducen la labor de crear un programa portable. Entre las tareas simplificadas con el uso de Autotool están:

1. Construcción de paquetes de software multidirectorios. Esto resulta mucho más difícil si se emplea *make* recursivamente. Simplificando este paso se estimula al desarrollador a organizar mediante un árbol de directorio todo el código fuente.

2. Configuración automática. No es necesario editar manualmente los archivos *makefile* cuando se desarrollen versiones de una aplicación para diferentes sistemas.
3. Generación automática de *Makefile*. Escribir ficheros tipo *Makefiles* implica muchas repeticiones y en proyectos muy extensos es muy probable la ocurrencia de errores. El sistema de construcción GNU genera ficheros tipo *Makefiles* automáticamente, solo se requiere escribir *makefile.am*, archivo más fácil de mantener.
4. Librerías compartidas. La construcción de librerías compartidas es tan sencilla como la construcción de librerías estáticas.

Cualquier proyecto de Autotool precisa dos archivos:

- *Makefile.am*: Archivo de entrada para Automake que incluye los requerimientos de construcción del proyecto.
- *Configure.ac*: Archivo de entrada para Autoconf que proporciona invocación de macros y fragmentos de código Shell utilizado para construir un script llamado *configure*.

El resto de los archivos necesarios para construir el proyecto son generados automáticamente.

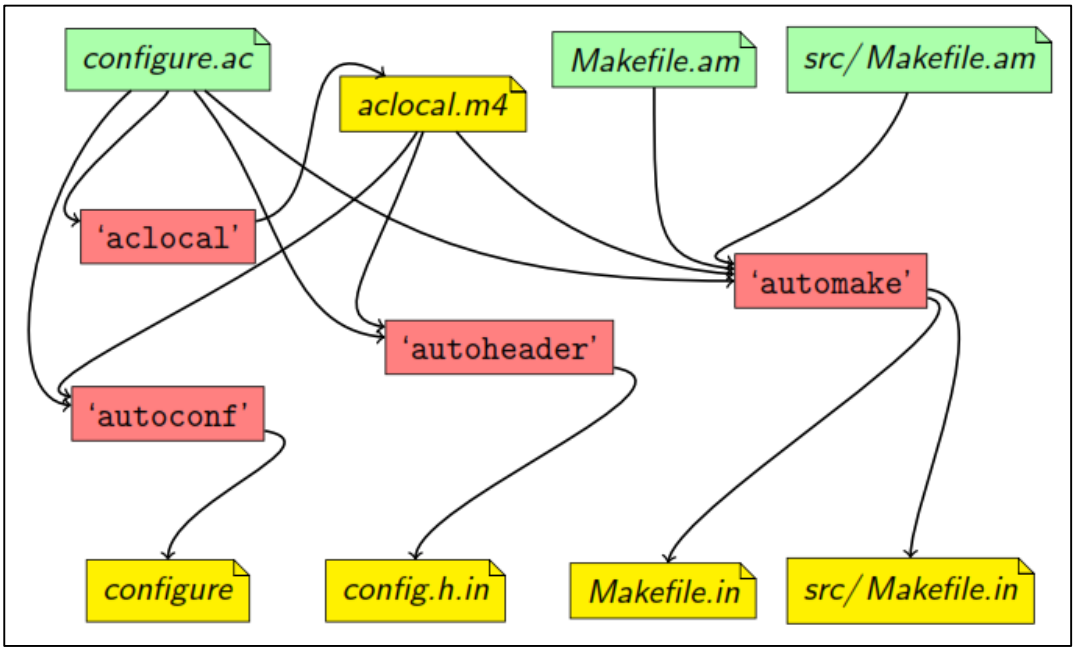

Ilustración 9 Generación de todos los archivos plantilla

2.7 Analizadores de Rendimiento

Los analizadores de rendimiento (denominados comúnmente profilers en inglés), son herramientas que permiten analizar el comportamiento de programas en tiempo de ejecución. Dicho análisis permite observar cuales son las partes del programa que consumen más tiempo de ejecución, y así saber cuáles zonas son críticas para optimizarlas.

Este tipo de herramientas permiten crear programas más eficientes a raíz de aspectos como son: el descubrir partes no utilizadas del código fuente, cuales son las secciones de la aplicación utilizadas con mayor frecuencia y en que funciones se invierte la mayor parte del tiempo de computo en la CPU.

Dentro de estas herramientas destaca gprof, la cual se puede acceder mediante el compilador gcc; el cual sumado a su gran potencia y versatilidad se destaca por ser software de código abierto, lo que lo hace totalmente gratuito.

La herramienta gprof nos proporciona perfiles de la ejecución de un programa, es decir, nos da información acerca de cuánto tiempo se emplea en cada función y de cuántas veces se llama. Nos facilita detectar dónde el programa está invirtiendo la mayor parte de su tiempo de ejecución.

La forma en que estas herramientas permiten al usuario conocer este tipo de comportamiento en su aplicación es mediante la generación de un fichero externo con información detallada del perfil de su programa; como lo es el caso de gmon.out que se analiza con gprof. Mediante este tipo de perfilado de programas se puede obtener información acerca de la distribución del tiempo de ejecución entre sus funciones y el número de veces que fueron llamadas.

Además hay que diferenciar que dependiendo de la herramienta o de los parámetros que se le ingresen, este fichero tendrá un flujo con los eventos grabados (rastro o grafo de llamada) o un resumen estadístico de los eventos observados (perfil plano). Los perfiladores utilizan una gran variedad de técnicas para recolectar información, incluyendo interrupciones de hardware, instrumentación de código y contadores de rendimiento. Esta diferencia crea los dos grandes grupos de *profilers*: los llamados perfiladores planos (*Flat profiler* en inglés) y los Perfiladores con Grafo de llamadas (*Call-Graph profiler* en inglés).

La salida producida por gprof se estructura en tres partes. La primera conocida como perfil plano, en el cual encontramos la información del tiempo consumido por cada función. Como segunda salida encontramos al grafo de llamadas, el cual nos muestra el análisis del tiempo de ejecución de cada función desde su invocación inicial hasta que culmine toda su instrucción, tiempo durante el cual estuvo el proceso en ejecución. En este análisis se toma en cuenta qué funciones fueron las que realizaron la invocación y cuáles fueron invocadas desde esta. Y como tercera salida aparece un listado alfabético de las funciones numeradas, para poder comprender mejor el grafo de llamadas (Graham et al., 1982).

Perfil plano:

Muestra información del tiempo consumido por cada función sin entrar en detalle en el grafo de llamadas. Esta salida es una tabla en la que se muestra una lista de rutinas ordenadas de mayor a menor según el consumo de la CPU. La misma tiene las siguientes columnas e información:

• %time: es un porcentaje del tiempo total consumido en la función, sin tomar en cuenta los llamados hechos a funciones externas.

• cumulative seconds: es el acumulado del tiempo total en segundos de esta función sumado con las funciones que aparecen listadas en la tabla antes de ella. No incluye el llamado a otras funciones.

• self seconds: es el tiempo en segundos consumidos por la función, sin tomar en cuenta llamados a otras funciones.

• calls: número de veces que se llama a la función.

• self us/ call: tiempo promedio en microsegundos consumido en llamados a la función. Sin tomar en cuenta llamados a otras funciones.

• total us/ call: tiempo promedio en microsegundos consumido en llamados a la función.

• name: nombre de la función.

De estar interesado únicamente en el perfil plano se debe de ejecutar el programa gprof únicamente con el parámetro -p.

Grafo de llamadas:

Existen casos en los que el uso del perfil plano no nos muestra toda la información que necesitamos conocer para poder realizar las mejoras que buscamos en nuestro código, en estos casos gprof nos presenta un grafo de llamadas en el cual podemos saber que funciones fueron invocadas por determinada función, o bien conocer que funciones son las responsables de invocar una función específica. De igual forma puede ser relevante conocer el tiempo que tardan en realizarse las llamadas a otras funciones. Esta información se nos muestra en forma de tabla con la siguiente información de la función en cada columna:

• index: es el índice que se le asigna. Por encima de ella se encuentran las funciones que la invocan y por debajo se encuentran las que son llamadas por esta.

• %time: es el porcentaje del tiempo de ejecución que utiliza, junto con los llamados a funciones desde la misma.

• self: es el tiempo en segundos consumido por la función, sin tomar en cuenta llamados a otras funciones.

• children: es el tiempo en segundos consumido por las funciones que son invocadas por esta.

• called: indica el número de veces que se invoca a la función que se analiza en la tabla, para el caso de las filas superiores. Para el caso de la función actual se muestra la cantidad de veces que ha sido llamada. Y para las filas inferiores muestra la cantidad de veces que se han llamado desde la función actual.

• name: nombre junto con el índice que se le asigna.

Modo de empleo de gprof:

Compilar con la opción -pg y sin optimizar.

Ejecutar el programa con un caso “típico”, esto crea el fichero gmon.out.

Ejecutar gprof nombre_ejecutable

Opciones:

-p muestra el perfil plano de la ejecución.

-q muestra el grafo de llamadas.

-b no muestra la explicación de los perfiles.

Un problema que presenta gprof en kernels como Linux es que no funciona correctamente con aplicaciones multihilos, perfilando solo el hilo principal.

Gprof emplea el *timer* interno ITIMER_PROF el cual permite al kernel enviar una señal a la aplicación cada vez que termine. Como solución a la situación problémica anterior solo se necesita pasar los datos de este *timer* a cada hilo creado. Con este fin se implementa un encapsulado para la función pthread_create en un archivo fuente llamado *multihilo_gprof.c* que se muestra en el ANEXO A (Hocevar, 2004).

2.8 Evaluación del software

SLOCCount

Con el objetivo de evaluar el software y realizar una comparativa entre las versiones de la aplicación se emplea la utilidad SLOCCount de código abierto. Este programa permite medir el tamaño de un programa en líneas de código fuente (SLOC por sus siglas en inglés) físicas, así como la estimación según el modelo básico COCOMO del esfuerzo de desarrollo (en personas-año y personas-mes), tiempo de desarrollo, número medio de desarrolladores y el coste total del proyecto.

Se ejecuta pasándole como parámetro el directorio que contiene el código fuente, aunque hay otros parámetros útiles, como *-personcost* para indicar el sueldo anual de un programador, cuyo valor por defecto es de $56,286 al año.

COCOMO

El Modelo de Costo Constructivo (o COCOMO, por su acrónimo del inglés COnstructive COst MOdel) es un modelo matemático de base empírica utilizado para estimación de costos de software. Incluye tres submodelos, cada uno ofrece un nivel de detalle y aproximación, cada vez mayor, a medida que avanza el proceso de desarrollo del software: básico, intermedio y detallado.

Pertenece a la categoría de modelos de subestimaciones basados en estimaciones matemáticas. Está orientado a la magnitud del producto final, midiendo el tamaño del proyecto en líneas de código principalmente (Boehm et al., 2000).

2.9 Generación de Documentación

Documentar el código de un programa es añadir suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores sepan qué hacer, sino que además los humanos entiendan qué están haciendo y por qué.

Un sistema pobremente documentado carece de valor aunque haya funcionado bien en alguna ocasión. La mayoría de los programas cuya única documentación es el código, se quedan obsoletos rápidamente y es imposible mantenerlos.

Existen multitud de razones válidas para escribir documentación en un proyecto de software:

- Es un requisito en un proyecto comercial, donde el cliente solicita que se documenten distintos componentes o fases del proyecto.
- Se emplea cuando una aplicación necesita interactuar con bases de datos externas u otras aplicaciones.
- Es necesario cuando varios grupos trabajan en distintos componentes de un mismo sistema.
- Permite la comunicación con equipos externos.
- Permite consolidar el conocimiento y aumentar la comprensión de un problema.

Se elige como generador de documentación a Doxygen teniendo en cuenta las características que presenta, entre las que se encuentran: compatibilidad con el lenguaje de programación C, es gratis, de código abierto con licencia GPL y es fácil de utilizar. Además, crea la documentación separada en listas y jerarquía de clases, por archivos, módulos o estructuras de datos. Ofrece la posibilidad de generar archivos en varios formatos:

• HTML
• LATEX
• RTF
• Postscript, PDF
• Página man de UNIX
• HTML comprimido de ayuda para Windows
• XML

Doxygen es un documentador automático capaz de extraer la información del propio código fuente. Está basado en la conocida herramienta Javadoc.

Para generar documentación adecuada debemos editar una plantilla de configuración con los datos más importantes del proyecto. Algunos de estos son:

PROJECT_NAME	Indica el nombre del proyecto sin espacios intermedios
PROJECT_NUMBER	Especifica la versión del programa.
INPUT	Lista de los nombres de archivos fuente.
FILE_PATTERNS	Extensiones u otro tipo de patrón de nombres, de los archivos a documentar.
EXTRACT_ALL	Documenta un programa que no tiene los estilos de comentario de Doxygen.
SOURCE_BROWSER	Se genera una documentación que hace referencia (tiene hiperenlaces) a una copia del código fuente.
OUTPUT_DIRECTORY	Especifica el directorio de salida; que puede ser absoluto o relativo.

Para que Doxygen interprete correctamente el código comentado se debe indicar de un modo adecuado lo que queremos expresar, empleando el formato que normalmente se ha utilizado en C y la inclusión en los comentarios de una serie de etiquetas especiales interpretadas por el programa para la codificación de la información.

Las etiquetas más comunes son:

@param	Información sobre un parámetro de una función.
@return	Información sobre lo que devuelve una función.
@brief	Explicación muy breve sobre lo que hace una función, de manera que no sea necesario verse toda la documentación en detalle. Aparece en el índice general.
@note	Aviso que aparece resaltado en la documentación de una función.
@see	Permite establecer referencias cruzadas con otras entidades.

Cuando en el archivo de configuración de Doxygen se incluye la opción PREDEFINED para definir una macro, Doxygen la utiliza luego para generar la documentación. Utilizando esta habilidad es posible definir un bloque de documentación para cada lenguaje, poniendo el texto de la documentación encerrado dentro de una pareja de #ifdef - #endif que corresponda a ese lenguaje.

2.10 Conclusiones parciales del capítulo

El desarrollo de mejoras y la corrección de errores a un proyecto de software es un proceso tedioso y meramente práctico que principalmente ofrece resultados cuando se prueba una versión de la aplicación que se construye. La mejor forma de entender la implementación de un programa es estudiar la documentación que se posea del mismo, la obtención de buenos resultados en esta labor depende del esfuerzo realizado por su desarrollador o desarrolladores en plasmar con sencillez y certeza el funcionamiento del código y la versatilidad de la herramienta que utilice. La inserción de comentarios en forma de documentación aprovechando las ventajas que ofrece Doxygen es una tarea que eleva el valor del proyecto y mejora el acabado del software.

La implementación está caracterizada por una política de empleo de código fuente abierto. Esto determina que el proyecto se siga considerando desde el punto de vista del software como una aplicación de bajo presupuesto y permita la integración de otros participantes en el proceso de perfeccionamiento y optimización que se viene llevando a cabo por el grupo GARP.

El entorno de ejecución es un elemento clave que conviene adaptar al tamaño y complejidad de las necesidades del sistema. Se debe tener en cuenta su efecto en los tiempos de respuesta y su correcta selección brinda grandes funcionalidades a la hora de la implementación.

El sistema de construcción empleado “GNU Build System” no sólo mejora la portabilidad, sino que simplifica la especificación de la construcción de las aplicaciones.

CAPÍTULO 3. RESULTADOS Y ANÁLISIS

3.1 Introducción

En los capítulos anteriores se expusieron los elementos de carácter teórico que anteceden a esta investigación, así como una explicación de herramientas seleccionadas y métodos empleados para el desarrollo de la implementación. En este capítulo se presenta la discusión de los resultados obtenidos mediante pruebas experimentales sobre el producto y el análisis de los datos recopilados.

3.2 Perfilado y requisitos de tiempo real

La verificación del correcto funcionamiento del software se realiza en condiciones de laboratorio con el autopiloto desmontado del vehículo y conectados la IMU, el módulo supervisor y el de control. El tiempo de duración de cada una de las dos pruebas efectuadas es de 10 minutos, donde se mantienen en ejecución cada versión de la aplicación luego de haberlas configurado con la opción *CFLAGS=-pg* que permite registrar cada evento y llamada a función. El fichero *gmon.out* es procesado con la herramienta gprof y se obtiene información detallada del perfil del programa.

A partir del análisis de rendimiento de la versión inicial, donde se revela una cantidad de llamadas a la función THREAD_Start() (encargada de crear un nuevo hilo de proceso) que sobrepasa el valor esperado, se implementa una secuencia de creación de hilos de procesos como se muestra en la ilustración No. 10. La comparación realizada en el ANEXO B ejemplifica como se logra disminuir a 3 ocurrencias la cantidad de llamadas a la función THREAD_Start() luego de implementar las mejoras.

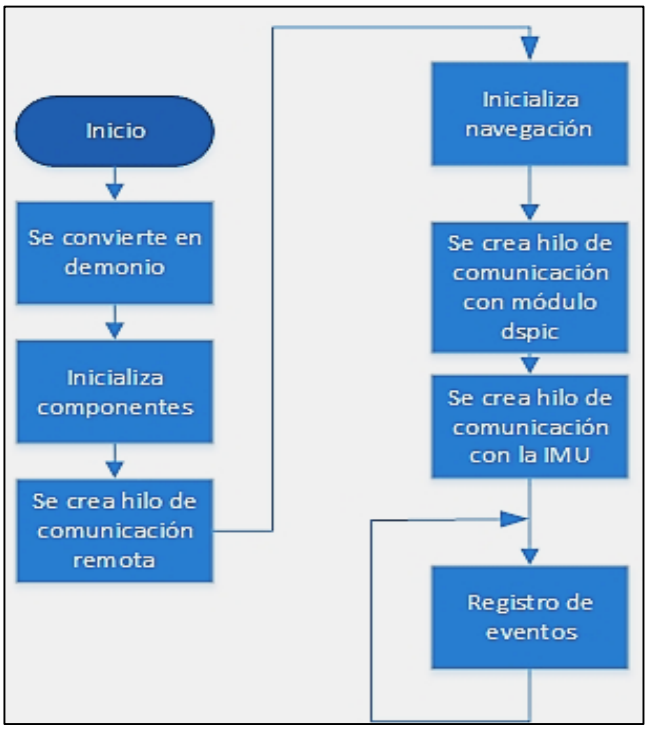

Ilustración 10 Secuencia de creación de hilos

Con el objetivo de garantizar los periodos de tiempo requeridos en cada ciclo se utiliza el algoritmo que se aprecia en la ilustración No. 11. Para evaluar la efectividad de los mismos se emplearon datos obtenidos de un experimento real efectuado en la tercera semana de diciembre del 2013, del cual se obtuvo una población de 519938 muestras.

En la Base de Datos Históricos se guarda la marca de tiempo del momento en que se almacenan los datos en cada ciclo. Con esta información es posible determinar el periodo de la tarea de almacenamiento. Para ello se procesa el fichero resultante de cada una de las pruebas realizadas durante el experimento, se extrae la marca de tiempo de cada registro de datos y a partir de estas se determina en cada caso el periodo de la tarea, restando cada marca de tiempo de la anterior. En la ilustración No. 12 se muestra la distribución del periodo de muestreo de esta tarea para la versión actual, con una media aritmética (μ) de 10.315ms y

una desviación estándar (σ) de 0.011012 y el comportamiento de los periodos de la misma tarea en la versión anterior.

Una medida de la calidad de los resultados ofrecidos arriba se obtiene al confrontar estos datos con los obtenidos en (Kim et al., 2012), donde se expone una prueba del cumplimiento de los requisitos de tiempo real en dos Sistemas Operativos (Microsoft Windows y Linux). Este experimento arroja que la instalación empleada del sistema operativo Linux fue capaz de lograr planificar una tarea con un periodo de 10ms, lográndolo con un error promedio de temporización de 0.054ms, lo cual valida los resultados presentados anteriormente.

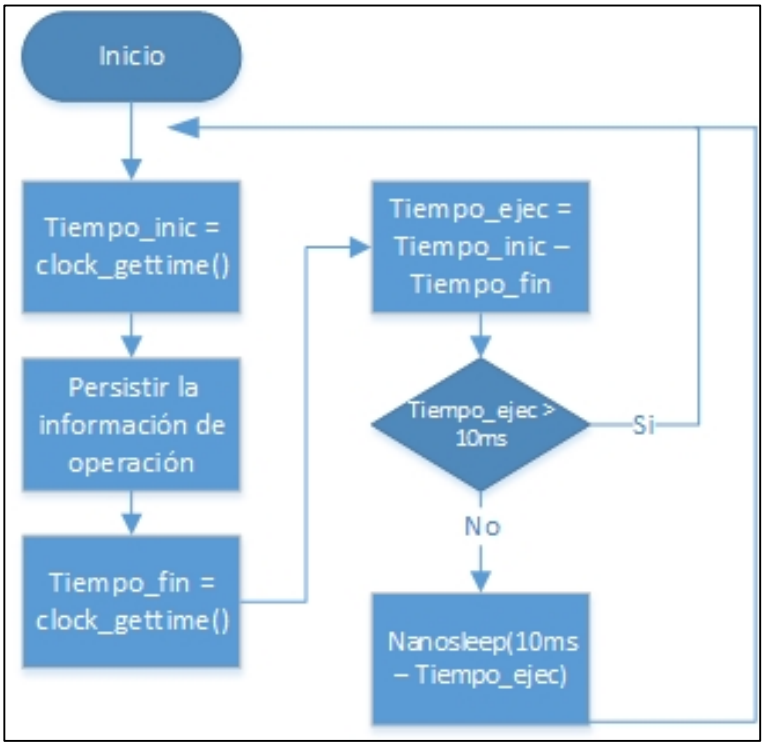

Ilustración 11 Algoritmo que garantiza el periodo de almacenamiento de datos históricos

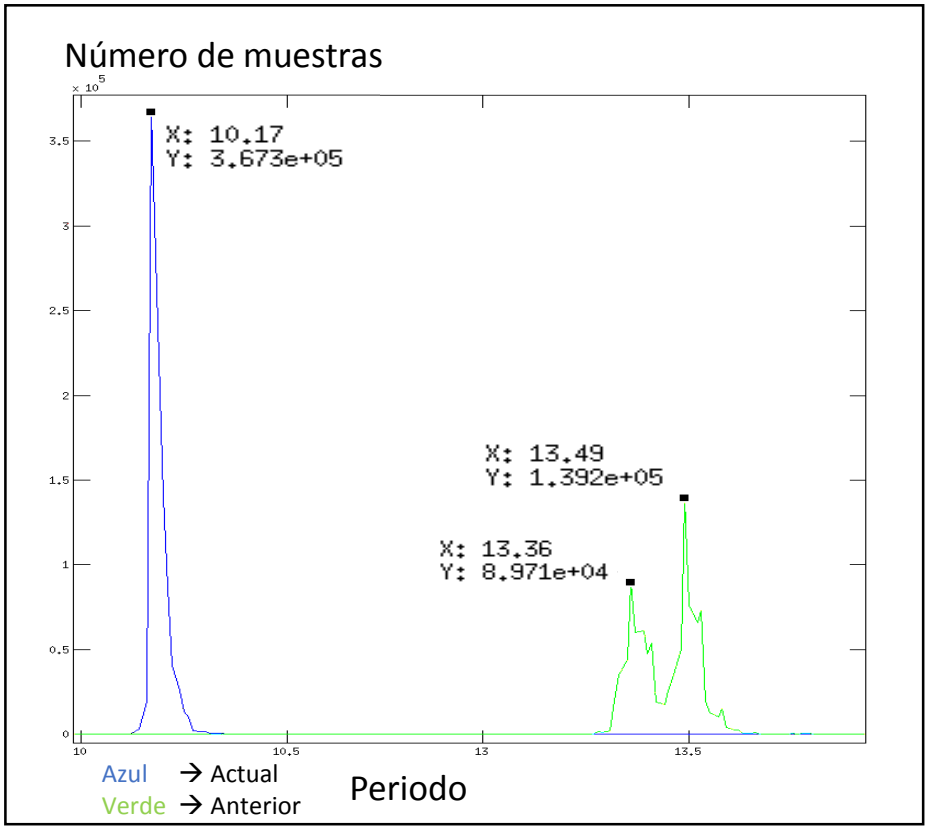

Ilustración 12 Comparación del periodo de muestreo de datos desde la IMU entre versiones de la aplicación

3.3 Análisis de *logs*

Producto de la implementación de syslog para crear un histórico de eventos, se obtiene un fichero *daemon.log* que puede ser analizado mediante el programa *glogg*. El fichero utilizado recoge información de experimentos reales efectuados en los meses de febrero y marzo del año 2014. En la ilustración No. 13 se pueden apreciar una muestra de los informes de errores en la comunicación con la IMU, la fecha y hora de archivo.

Esta funcionalidad ofrece la ventaja de registrar errores y eventos durante el transcurso de la misión, permite conocer la hora de iniciado el sistema y el tiempo que estuvo activo así como registrar los parámetros del destino cuando el sistema está en seguimiento de trayectoria, beneficios de los que no se disponía anteriormente.

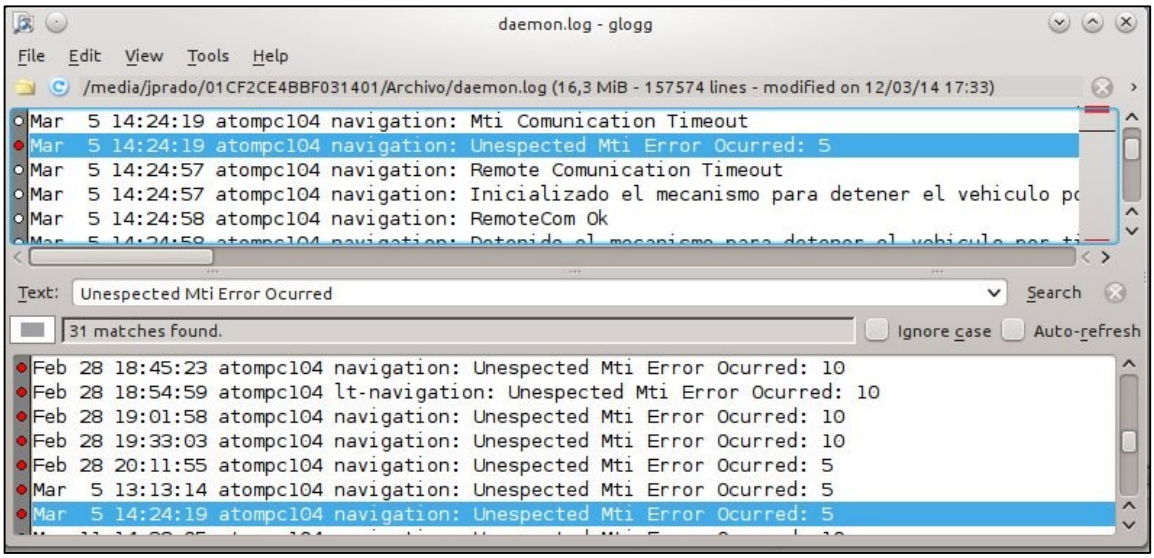

Ilustración 13 Análisis de *logs*

En la tabla 3.1 se resumen las fallas encontradas en un total de 33 horas y 38 minutos con 42 segundos aproximadamente. Los 54 errores de comunicación con la IMU representan un aproximado de 0.0004 % de ocurrencia teniendo en cuenta que el ciclo de ejecución es de 10ms, por lo que se considera que estos no son producto de dificultades en el software. Son probables causas: ruidos inducidos por las elevadas corrientes que maneja el motor del vehículo en los casos de fallo en el chequeo de suma del protocolo de comunicación y problemas de hardware en la MTI-G para los errores desconocidos. Para los 53 errores en la comunicación con el supervisorio que representan un 0.0043 % de ocurrencia considerando el periodo de atención a supervisorio de 100ms, se evalúa la posibilidad de que sean originados por inconvenientes en el hardware del modem transmisor.

Error	**Ocurrencia**
MTI (desconocido)	5
MTI (chequeo de suma)	49
Comunicación remota	53
Total de horas Activo	33h 38min 42s

Tabla 3.1 Errores encontrados

3.4 Evaluación del software

Evaluar un software es una tarea muy difícil para la cual se deben tomar en cuenta muchos factores tanto económicos como del impacto de su utilización. La herramienta SLOCCount ofrece los resultados plasmados en la tabla 3.2, resaltando una disminución en la cantidad de líneas de código, que aunque puede parecer contradictorio, es válido aclarar que se eliminaron más segmentos de código innecesarios que los agregados para resolver las deficiencias. El costo estimado también se redujo en un valor de $ 20 000, lo que confirma el carácter de bajo costo del proyecto y una mayor simplicidad en el código.

Características	**Antes**	**Ahora**
Total de líneas de código	4 844	4 105
Estimado cantidad de desarrolladores	1,92	1,73
Estimado tiempo planificado (meses)	6,54	6,13
Costo total de proyecto (salario promedio al año $ 56286)	$ 141 614	$ 119 020

Tabla 3.2 Análisis de costo

3.5 Documentación

Los comentarios realizados al código en forma de documentación no determinan si el programa funciona correctamente o de forma más eficiente, pero eleva el valor del proyecto haciéndolo más fácil de comprender y de mantener para sus desarrolladores. También aumenta la calidad del software como producto final y agiliza las consultas por personal ajeno a su producción.

Como norma para el desarrollo de esta tarea se considera de mayor prioridad la documentación del nuevo código introducido. Estos módulos son: include/checksum.h, include/garpcom.h, include/dspic_comunication.h, include/dspic_com_constant.h,

include/remote_comunication.h, include/remote_com_constant.h, include/mti_comunication.h, include/mti_com_constant.h, include/piti2.0.h.

En la ilustración No. 14 se muestra un ejemplo de una función que se encuentra comentada utilizando las etiquetas de Doxygen. En este caso se puede observar como el nombre de dicha función no ofrece información de su utilidad y los parámetros tienen características desconocidas. Un fragmento del archivo HTML generado por Doxygen se expone en la ilustración No. 15 donde se explica de forma precisa el funcionamiento de dtfunction_apply().

```
/** Calcula un nuevo resultado empleando la función de transferencia discreta.
 *
 * @param dtf Puntero a la variable que almacena la función transferencial.
 * @param input Valor de entrada.
 * @return Valor resultado.
 *
 * @ingroup dtfunction
 */
double dtfunction_apply(struct dtfunction * dtf, double input);
```

Ilustración 14 Función documentada

Documentación de las funciones

double dtfunction_apply (struct dtfunction * dtf,
double input
)

Calcula un nuevo resultado empleando la función de transferencia discreta.

Parámetros

dtf Puntero a la variable que almacena la función transferencial.

input Valor de entrada.

Devuelve

Valor resultado.

Ilustración 15 Fragmento del html generado como documentación

3.6 Sistema de Construcción

Se realizaron pruebas de portabilidad hacia otros sistemas como: Ubuntu, Fedora, Open BSD, Debian y Mac OS X. Como resultado se obtuvo que la aplicación es portable a otros Sistemas Operativos basados en el núcleo de Linux como Debian y Fedora en situaciones donde se cumplan los requerimientos del ambiente de ejecución tanto de hardware como software. Sin embargo, los intentos de portabilidad hacia otros sistemas como OpenBSD y Mac OS X fallaron al compilar *src/timer.c., r*equiriendo este último que el sistema sea compatible con la extensión de POSIX de tiempo real (POSIX.1003.1b, 1996).

Los resultados demuestran que el sistema de construcción no es el único elemento que impacta en la portabilidad, siendo necesario además que el código sea portable. Aunque no se alcanza el objetivo de portar la aplicación a todas las plataformas anteriormente mencionadas, se considera un logro, debido a que, con la versión anterior no hubiese sido posible compilar el proyecto en ninguna de estas.

3.7 Conclusiones parciales del capítulo

En este capítulo se analizan los datos recopilados en varios experimentos reales y se prueba el rendimiento de la aplicación. A partir de los resultados obtenidos se concluye que:

- Los algoritmos empleados en la implementación de soluciones permiten la reducción del número de hilos creados en tiempo de ejecución y el cumplimiento de los requisitos de tiempo real blando en la tarea de adquisición de datos desde la IMU y su transmisión hacia el sistema de control.
- El análisis de la información de depurado permite identificar causas de problemas existentes en el sistema.
- La selección del “Sistema de Construcción de GNU” como fuente de portabilidad no es el único elemento a considerar en la producción de software portable hacia otras plataformas.

CONCLUSIONES Y RECOMENDACIONES

CONCLUSIONES

- Es posible cumplir con los requerimientos de tiempo real blando impuestos al software de nivel táctico del HRC-AUV empleando un Sistema Operativo basado en el núcleo de Linux con las características de hardware antes mencionadas.
- El análisis de la información de depurado permite identificar causas de problemas existentes en el sistema.
- La implementación de mejoras y funcionalidades realizada propicia que la nueva versión de la aplicación presente un mejor desempeño en ejecución.
- La aplicación fue probada de forma satisfactoria cumpliendo los requerimientos de tiempo real blando en la tarea de adquisición de datos desde la Unidad de Mediciones Inerciales y su transmisión al sistema de control.

RECOMENDACIONES

- Realizar pruebas donde se induzcan fallos al hardware para detectar y corregir comportamientos inadecuados del software.
- Continuar la identificación de posibles mejoras en base a la experiencia de futuros experimentos reales.
- Estudiar la posibilidad de aplicar los conjuntos de herramientas basadas en metaOS dadas sus características de software libre y las amplias referencias encontradas en la bibliografía especializada.
- Aplicar la herramienta *gcov* para identificar áreas de código no utilizadas, pues su inclusión en sistemas críticos no está permitido en las normas internacionales.

REFERENCIAS BIBLIOGRÁFICAS

2004. The Navy Unmanned Undersea Vehicle (UUV) Master Plan. US Department of the Navy.

BARNEY, B. 2009. POSIX threads programming. *National Laboratory. Disponível em:< https://computing. llnl. gov/tutorials/pthreads/> Acesso em,* 5.

BOEHM, B. W., MADACHY, R. & STEECE, B. 2000. *Software Cost Estimation with Cocomo II with Cdrom*, Prentice Hall PTR.

BRUTZMAN, D., BURNS, M., CAMPBELL, M., DAVIS, D., HEALEYH, T., HOLDEN, M., LEONHARDT, B., MARCO, D., MCCLARIN, D., MCGHEE, B. & WHALEN, R. NPS Phoenix AUV software integration and in-water testing. , Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology, 1996. AUV '96, 1996 1996. 99-108.

BUTTAZZO, G. & LIPARI, G. Ptask: An educational C library for programming real-time systems on Linux. 2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA), Sept 2013 2013. 1-8.

CALCOTE, J. 2010. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*, No Starch Press.

CAÑIZARES, J. R. 2010. *Modelado y control del Vehículo Autónomo Sumergible del CIDNAV*. Universidad Central "Marta Abreu" de Las Villas.

CHEW, J. L., KOAY, T. B., TAN, Y. T., ENG, Y. H., GAO, R., CHITRE, M. & CHANDHAVARKAR, N. 2011. Starfish: An open-architecture auv and its applications. *Defense Technology Asia*.

DATARDINA, S. P., DU CROZ, J. J., HAMMARLING, S. J. & PONT, M. W. 1992. A proposed specification of BLAS routines in C. *The Journal of C Language Translation,* 3**,** 295–309.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S. & DUFF, I. S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS),* 16**,** 1–17.

FOSSEN, T. I. 2002. *Marine control systems: guidance, navigation and control of ships, rigs and underwater vehicles*, Marine Cybernetics Trondheim.

FOSTER, H., MAGEE, J., KRAMER, J. & UCHITEL, S. Adaptable software architectures and task synthesis for uavs. 2006 2006.

FUJII, T., URA, T. & KURODA, Y. Mission Execution Experiments with a Newly Developed AUV" The Twin-Burger". 1993 1993. 92–92.

GALASSI, M., DAVIES, J., THEILER, J., GOUGH, B., JUNGMAN, G., ALKEN, P., BOOTH, M. & ROSSI, F. 2009. GNU Scientific Library Reference Manual-(v1. 12). *Network Theory Ltd*.

GARCÍA GARCÍA, D. 2010. *Técnicas para el incremento de las prestaciones de los sistemas de navegación inercial de bajo costo para vehículos autónomos.* Universidad Central "Marta Abreu" de Las Villas.

GRAHAM, S. L., KESSLER, P. B. & MCKUSICK, M. K. Gprof: A call graph execution profiler. 1982 1982. ACM, 120–126.

GUERRA MORALES, C. E. G. 2010. *Diseño e implementación de hardware y software de bajo nivel para vehículo submarino autónomo.* Universidad Central "Marta Abreu" de Las Villas.

HOCEVAR, S. 2004. HOWTO: using gprof with multi-threaded applications. *Available from World Wide Web: http://sam.zoy. org/writings/programming/gprof. html*.

JOINT TECHNICAL COMMITTEE ISO/IEC JTC 1, I. T. 1999. International Standard ISO/IEC 9899. ISO Secretariat.

KIM, B., JUN, B.-H., PARK, J.-Y. & LEE, P.-M. Real-time process without RTOS for the ISiMI100 Autonomous Underwater Vehicle. OCEANS, 2012 - Yeosu, 2012 2012. 1-4.

KOPETZ, H. 2011. *Real-time systems: design principles for distributed embedded applications*, Springer.

KRAETZSCHMAR, G. K., SHAKHIMARDANOV, A., PAULUS, J., HOCHGESCHWENDER, N. & RECKHAUS, M. 2010. Best Practice in Robotics.

KWAK, S.-H., MCGHEE, R. B. & BIHARI, T. E. 1992. Rational Behavior Model: A Tri-Level Multiple Paradigm Architecture for Robot Vehicle Control Software. DTIC Document.

LAWSON, C. L. Background, Motivation and a Retrospective View of the BLAS., 1999 1999.

LEE, G.-M., PARK, J.-Y., KIM, B., BAEK, H., PARK, S., SHIM, H., CHOI, G., KIM, B.-R., KANG, H.-G. & JUN, B.-H. 2013. Development of an Autonomous Underwater Vehicle ISiMI6000 for Deep-sea Observation. *IJMS,* 42**,** 1034–1041.

LEMUS RAMOS, J. L. 2011. *Software de navegación y guiado en tiempo real para Vehículo Autónomo Sumergible.* Universidad Central "Marta Abreu" de Las Villas.

MADDEN, C. 2013. An Evaluation of Potential Operating Systems for Autonomous Underwater Vehicles.

MARTINEZ, A., RODRIGUEZ, Y., HERNANDEZ, L., GUERRA, C. & SAHLI, H. 2010. Hardware and software architecture for AUV based on low-cost sensors. 1428-1433.

MATOS, A. & CRUZ, N. 2009. MARES-navigation, control and on-board software. *on "Underwater vehicles", ISBN*, 978–953.

MUELLER, F. A Library Implementation of POSIX Threads under UNIX., 1993 1993. 29–42.

PALOMERAS, N., CARRERAS, M., RIDAO, P. & HERNANDEZ, E. Mission control system for dam inspection with an AUV. 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2006 2006. 2551-2556.

RAGO, S. A. & STEVENS, W. R. 2005. *Advanced programming in the UNIX Environment*, Addison-Wesley Professional Computing Series.

ROBERTS, R. L. 1997. Analysis, Experimental Evaluation and Software Upgrade for Attitude Estimation by the Shallow-Water AUV Navigation System (SANS).

SICILIANO, B. & KHATIB, O. 2008. *Springer handbook of robotics*, Springer.

STOKEY, R. P., ROUP, A., VON ALT, C., ALLEN, B., FORRESTER, N., AUSTIN, T., GOLDSBOROUGH, R., PURCELL, M., JAFFRE, F. & PACKARD, G. Development of the REMUS 600 autonomous underwater vehicle. 2005 2005. IEEE, 1301–1304.

WEST, M. E., NOVITZKY, M., VARNELL, J. P., MELIM, A., SEQUIN, E., TOLER, T. C., COLLINS, T. R. & BOGLE, J. R. 2010. Design and Development of the Yellowfin UUV for Homogenous Collaborative Missions.

ANEXO A. ENCAPSULADO PARA LA FUNCIÓN PTHREAD_CREATE

```
#define _GNU_SOURCE
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <pthread.h>

static void * wrapper_routine(void *);

static int (*pthread_create_orig)(pthread_t *__restrict,
                                  __const pthread_attr_t *__restrict,
                                  void *(*)(void *),
                                  void *__restrict) = NULL;

void wooinit(void) __attribute__((constructor));

void wooinit(void)
{
   pthread_create_orig = dlsym(RTLD_NEXT, "pthread_create");
   fprintf(stderr, "pthreads: using profiling hooks for gprof\n");
   if(pthread_create_orig == NULL)
   {
      char *error = dlerror();
      if(error == NULL)
      {
         error = "pthread_create is NULL";
      }
      fprintf(stderr, "%s\n", error);
      exit(EXIT_FAILURE);
```

```
    }
}

typedef struct wrapper_s
{
    void * (*start_routine)(void *);
    void * arg;

    pthread_mutex_t lock;
    pthread_cond_t  wait;

    struct itimerval itimer;

} wrapper_t;

static void * wrapper_routine(void * data)
{
    void * (*start_routine)(void *) = ((wrapper_t*)data)->start_routine;
    void * arg = ((wrapper_t*)data)->arg;

    setitimer(ITIMER_PROF, &((wrapper_t*)data)->itimer, NULL);

    pthread_mutex_lock(&((wrapper_t*)data)->lock);
    pthread_cond_signal(&((wrapper_t*)data)->wait);
    pthread_mutex_unlock(&((wrapper_t*)data)->lock);

    return start_routine(arg);
}

int pthread_create(pthread_t *__restrict thread,
                   __const pthread_attr_t *__restrict attr,
                   void * (*start_routine)(void *),
                   void *__restrict arg)
{
    wrapper_t wrapper_data;
    int i_return;

    wrapper_data.start_routine = start_routine;
    wrapper_data.arg = arg;
    getitimer(ITIMER_PROF, &wrapper_data.itimer);
    pthread_cond_init(&wrapper_data.wait, NULL);
```

```
    pthread_mutex_init(&wrapper_data.lock, NULL);
    pthread_mutex_lock(&wrapper_data.lock);

    i_return = pthread_create_orig(thread,
                                   attr,
                                   &wrapper_routine,
                                   &wrapper_data);


    if(i_return == 0)
    {
       pthread_cond_wait(&wrapper_data.wait, &wrapper_data.lock);
    }

    pthread_mutex_unlock(&wrapper_data.lock);
    pthread_mutex_destroy(&wrapper_data.lock);
    pthread_cond_destroy(&wrapper_data.wait);

    return i_return;
}
```

ANEXO B. ANÁLISIS DE RENDIMIENTO

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
  1.45      0.69     0.01                             TIMER_Destroy
  0.00      0.69     0.00   203942     0.00     0.00  MUTEX_Lock
  0.00      0.69     0.00   184180     0.00     0.00  Fill_timespec_from_ms
  0.00      0.69     0.00    16600     0.00     0.00  QUEUE_GetAt
  0.00      0.69     0.00     3696     0.00     0.00  MATRIX_SetAt
  0.00      0.69     0.00     2647     0.00     0.00  GARPCOM_PushFloat
  0.00      0.69     0.00     1670     0.00     0.14  Display
  0.00      0.69     0.00     1670     0.00     0.00  ms_sleep
  0.00      0.69     0.00     1152     0.00     0.00  MTICOM_GetFloat
  0.00      0.69     0.00      576     0.00     0.00  FILTER_Aplu
  0.00      0.69     0.00      533     0.00     0.00  THREAD_Start
  0.00      0.69     0.00      531     0.00     0.00  THREAD_Wait
  0.00      0.69     0.00      354     0.00     0.00  DSPICCOM_SetStatus
  0.00      0.69     0.00      353     0.00     0.00  CalculateCheckSum
  0.00      0.69     0.00      353     0.00     0.00  GARPCOM_SendMessage
```

Ilustración 16 Análisis con gprof de la versión anterior

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 0.00      0.06     0.00      740     0.00     0.00  REMOTECOM_SetStatus
 0.00      0.06     0.00      726     0.00     0.00  GetData
 0.00      0.06     0.00      717     0.00     0.00  SendData
 0.00      0.06     0.00       14     0.00     0.00  SetDataPower
 0.00      0.06     0.00       13     0.00     0.00  DSPICCOM_SetConfigData
 0.00      0.06     0.00        8     0.00     0.00  MUTEX_Create
 0.00      0.06     0.00        5     0.00     0.00  SetDataRemoteManagement
 0.00      0.06     0.00        3     0.00     0.00  PORT_Create
 0.00      0.06     0.00        3     0.00     0.00  THREAD_Start
 0.00      0.06     0.00        1     0.00     0.00  DSPICCOM_Init
 0.00      0.06     0.00        1     0.00     0.00  HDB_Clear
```

Ilustración 17 Análisis con gprof de la versión actual

ANEXO C. DAEMON.C

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <stdarg.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/resource.h>

#define MAXLINE 4096            /* max line length */

static void err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
        char buf[MAXLINE];
        vsnprintf(buf, MAXLINE, fmt, ap);
        if (errnoflag)
                snprintf(buf + strlen(buf), MAXLINE - strlen(buf), ": %s",
                        strerror(error));
        strcat(buf, "\n");
        fflush(stdout); /* in case stdout and stderr are the same */
        fputs(buf, stderr);
        fflush(NULL); /* flushes all stdio output streams */
}

/*
 * Fatal error unrelated to a system call.
```

```
 * Print a message and terminate.
 */
void err_quit(const char *fmt, ...)
{
        va_list ap;

        va_start(ap, fmt);
        err_doit(0, 0, fmt, ap);
        va_end(ap);
        exit(1);
}

int lockfile(int fd)
{
        struct flock fl;

        fl.l_type = F_WRLCK;
        fl.l_start = 0;
        fl.l_whence = SEEK_SET;
        fl.l_len = 0;
        return (fcntl(fd, F_SETLK, &fl));
}

/* example settings for already_running()
 const char * LOCKFILE = "/var/run/navigation.pid";
 */
int already_running(const char * lock_filename)
{
        int fd;
        char buf[16];

        fd = open(lock_filename, O_RDWR | O_CREAT,
                  (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH));
        if (fd < 0) {
                syslog(LOG_ERR, "can't open %s: %s", lock_filename,
                        strerror(errno));
                exit(1);
        }
        if (lockfile(fd) < 0) {
                if (errno == EACCES || errno == EAGAIN) {
                        close(fd);
```

```
                return (1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", lock_filename,
                strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long) getpid());
    write(fd, buf, strlen(buf) + 1);
    return (0);
}

void daemonize(const char *cmd)
{
    int i, fd0, fd1, fd2;
    pid_t pid;
    struct rlimit rl;
    struct sigaction sa;
    /*
     * Clear file creation mask.
     */
    umask(0);

    /*
     * Get maximum number of file descriptors.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: can't get file limit", cmd);

    /*
     * Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);
    setsid();

    /*
     * Ensure future opens won't allocate controlling TTYs.
     */
```

```
        sa.sa_handler = SIG_IGN;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        if (sigaction(SIGHUP, &sa, NULL) < 0)
                err_quit("%s: can't ignore SIGHUP");
        if ((pid = fork()) < 0)
                err_quit("%s: can't fork", cmd);
        else if (pid != 0) /* parent */
                exit(0);

        /* Change the current working directory to the root so
         * we won't prevent file systems from being unmounted.
         */
        if (chdir("/") < 0)
                err_quit("%s: can't change directory to /");

        /*
         * Close all open file descriptors.
         */
        if (rl.rlim_max == RLIM_INFINITY)
                rl.rlim_max = 1024;
        for (i = 0; i < rl.rlim_max; i++)
                close(i);

        /*
         * Attach file descriptors 0, 1, and 2 to /dev/null.
         */
        fd0 = open("/dev/null", O_RDWR);
        fd1 = dup(0);
        fd2 = dup(0);

        /*
         * Initialize the log file.
         */
        openlog(cmd, LOG_CONS, LOG_DAEMON);
        if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
                syslog(LOG_ERR, "unexpected file descriptors %d %d %d", fd0,
                        fd1, fd2);
                exit(1);
        }
}
```