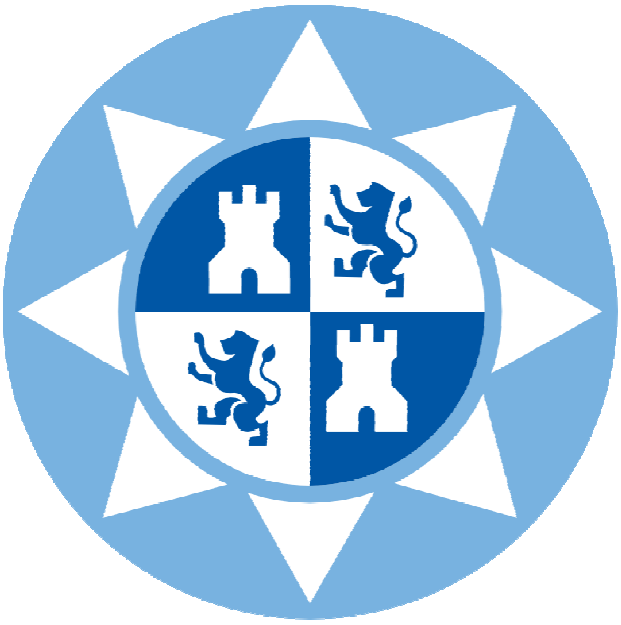
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN.
UNIVERSIDAD POLITÉCNICA DE CARTAGENA.



Máster de Tecnologías de la Información y Comunicaciones.

Control de vehículos con CompactRIO. Integración de aplicaciones Matlab y Java.

Autor: Ramón Martínez Carreras.
Director: Juan Ángel Pastor Franco.

09/2012

Autor	Ramón Martínez Carreras
E-mail del Autor	rmartinezcarreras@gmail.com
Director	Juan Ángel Pastor Franco
E-mail del Director	Juanangel.pastor@upct.es
Título del PFM	Control de vehículos con CompactRIO. Integración de aplicaciones MatLab y Java.
Descriptores	Ingeniería de software. Patrones de diseño. Autómatas programables.

Resumen

Se trata de definir un protocolo de comunicaciones basado en cadenas de caracteres entre una unidad de control CompactRIO y una aplicación Java.

Este protocolo debe permitir la integración de aplicaciones gráficas de tele-operación de vehículos implementadas en Java con software de control para MATLAB ejecutándose sobre una unidad CompactRIO.

Fases del Proyecto:

1. Estudio de toolboxes de comunicaciones de Matlab.
2. Estudio de toolboxes de generación XML en Matlab.
3. Estudio de librerías de generación XML en Java.
4. Pruebas de comunicaciones entre proceso MatLab y aplicación Java.
5. Definición de una aplicación de prueba consistente en un simulador MatLab y una aplicación Java.
6. Definición y prueba de un protocolo para la aplicación de prueba.
7. Diseño e implementación de la aplicación de teleoperación de Java.
8. Elaboración de la memoria.

Titulación	Máster Universitario en Tecnologías de la Información y Comunicaciones
Departamento	Tecnologías de la Información y Comunicaciones
Fecha de Presentación	09/2012

Índice general

Índice de figuras

Índice de tablas

Capítulo 1. Introducción

1.1. Motivación

Este Proyecto Fin de Máster de las Tecnologías de la Información y Comunicaciones (Máster TIC) titulado "Control de vehículos con CompactRIO. Integración de aplicaciones MatLab y Java" nace debido a la inquietud por la investigación en controladores lógicos programables o PLCs (acrónimo de *Programmable Logic Controllers*) así como por la adquisición de un prototipo de vehículo con una unidad de control CompactRIO [1] por el grupo División de Sistemas e Ingeniería Electrónica (DSIE) de la Universidad Politécnica de Cartagena (UPCT).

El controlador CompactRIO empotrado en el prototipo permite su implementación bajo el entorno Matlab. Por esta razón la meta de este Proyecto Fin de Máster es el desarrollo de aplicaciones Java y Matlab que se comuniquen mediante la transmisión de comandos XML.

Una vez superada esta barrera se podrá establecer una plataforma de pruebas real para conseguir un SW final de teleoperación que permita realizar un control remoto eficiente del prototipo.

Cabe mencionar que se trata de una gran oportunidad para la investigación en varios campos:

- Controladores lógicos con unidad operativa.
- Ingeniería de Software, patrones de diseño.
- Integración de sistemas.

Para llevar a cabo el diseño, en primer lugar se han investigado toolboxes de comunicaciones y generación de XML en Matlab. En segundo lugar se han investigado librerías que permiten la generación de XML en Java. Después de realizar pruebas de comunicaciones entre los desarrollos Matlab y Java se ha ideado un simulador en Matlab basado en el gobierno de un sistema propulsor empleado en la industria naval. El siguiente paso ha sido proveer al simulador de un juego de instrucciones (comandos) dotándolo de un formato XML.

Como resultado de este Proyecto Fin de Máster se ha obtenido una aplicación Java de teleoperación del simulador. Ambas aplicaciones se comunican mediante el intercambio de comandos XML.

1.2. Objetivos

El principal objetivo de este Proyecto Fin de Máster es definir un protocolo de comunicaciones basado en cadenas de caracteres entre una unidad de control CompactRIO y una aplicación Java.

Además este protocolo debe permitir la integración de aplicaciones gráficas de tele-operación de vehículos implementadas en Java con software de control para MATLAB ejecutándose sobre una unidad CompactRIO.

Para alcanzar la meta propuesta es necesario cumplir con los siguientes hitos:

1. Estudio de toolboxes de comunicaciones de Matlab.
2. Estudio de toolboxes de generación XML de Matlab
3. Estudio de librerías de generación de XML en Java.
4. Pruebas de comunicaciones entre proceso MatLab y Aplicación Java.
5. Definición de una aplicación de prueba consistente en un simulador MatLab y una aplicación Java.
6. Definición y prueba de un protocolo para la aplicación de prueba.
7. Diseño e implementación de la aplicación de teleoperación de Java.
8. Elaboración de una memoria descriptiva del Proyecto realizado.

1.3. Organización y contenido del documento

A continuación se detalla el contenido de cada uno de los capítulos que forman parte de esta memoria.

- Capítulo 2: Contexto de desarrollo. Este capítulo sirve para mostrar al lector el entorno de trabajo en el que se ha desarrollado el proyecto. Por esta razón contiene un breve resumen de cada una de las herramientas *software* utilizadas para llevarlo a cabo. Además contiene una introducción teórica del principio de diseño aplicado en el *software* del proyecto.

- Capítulo 3: Fundamentos de los Controladores Lógicos. En este capítulo se estudian los conceptos generales de los Controladores Lógicos. A continuación se introducen los Autómatas Programables basados en computador señalando las características más importantes. Finalmente se aborda el Controlador de Automatización Programable CompactRIO analizando su arquitectura hardware así como el software de desarrollo suministrado por el fabricante National Instruments.

- Capítulo 4: Toolboxes Matlab. En esta parte del documento se han descrito las funcionalidades que proveen las toolboxes de Matlab Instrument Control y XML empleadas para desarrollar las comunicaciones y la generación de comandos XML en el código Matlab objeto de este proyecto.

- Capítulo 5: Descripción del simulador. En este fragmento de la memoria se describe la funcionalidad del Bow Thruster así como los comandos que modelan el Panel de Emergencia Remoto que lo gobierna. Sucesivamente se realiza el estudio de la implementación del simulador en Matlab prestando especial atención al desarrollo de la Interfaz Gráfica de Usuario gracias a la funcionalidad GUIDE.

- Capítulo 6: Principios de diseño . En este capítulo de la memoria se explica la forma en la que se ha aplicado el principio de diseño Objeto Activo en la API desarrollada en el proyecto.

- Capítulo 7: Paquetes y librerías. En esta parte de la memoria justificamos el diseño en niveles de la API desarrollada en el transcurso del proyecto. Además se describe el paquete java.net y la librería JDOM ya que han sido de gran ayuda tanto en la implementación asociada a las comunicaciones como en la referente a la generación de comandos XML en Java.

- Capítulo 8: Desarrollo e implementación. En este capítulo se analizan los fragmentos de código de las aplicaciones Java y Matlab desarrolladas que requieren de un análisis más detallado debido a la dificultad de su programación como es el caso de las comunicaciones entre ambas implementaciones así como de la generación/interpretación de comandos XML en ambos lenguajes.

- Capítulo 9: Manual de usuario. En esta parte del documento se proporciona un manual para realizar una correcta ejecución de las Interfaces Gráficas de usuario implementadas en Java y Matlab.

- Capítulo 10: Conclusiones y trabajos futuros. En este capítulo se realiza un amplio resumen del proyecto. En el apartado de conclusiones se analizan los resultados obtenidos. Para finalizar se proponen algunas líneas de investigación interesantes relacionadas con este PFM.

- Capítulo 11: Referencias. En este capítulo se indican las fuentes de información que han servido de base para llevar a cabo este Proyecto Fin de Máster.

Capítulo 2. Contexto de Desarrollo

2.1. Matlab

El nombre de Matlab proviene de la contracción de los términos *MATrix LABoratory* y fue inicialmente concebido para proporcionar fácil acceso a las librerías LINPACK y EISPACK, las cuales representan hoy en día dos de las librerías más importantes en computación y cálculo matricial. Se trata de un software matemático que ofrece un entorno de desarrollo integrado (IDE) con un lenguaje de programación propio (lenguaje M) y está disponible para las plataformas Unix, Windows y Mac OS.

Matlab es un entorno de computación y desarrollo de aplicaciones totalmente integrado orientado para llevar a cabo proyectos en donde se encuentren implicados elevados cálculos matemáticos y la visualización gráfica de los mismos. Matlab integra análisis numérico, cálculo matricial, proceso de señal y visualización gráfica en un entorno completo donde los problemas y sus soluciones son expresados del mismo modo en que se escribirían tradicionalmente, sin necesidad de hacer uso de la programación tradicional.

En los medios universitarios Matlab se ha convertido en una herramienta básica, tanto para los profesionales e investigadores de centros docentes, como una importante herramienta para el dictado de cursos universitarios, tales como sistemas e ingeniería de control, álgebra lineal, proceso digital de imagen, señal, etc. En el mundo industrial Matlab está siendo utilizado como herramienta de investigación para la resolución de complejos problemas planteados en la realización y aplicación de modelos matemáticos en ingeniería. Los usos más característicos de la herramienta se encuentran en áreas de computación y cálculo numérico tradicional, prototipaje algorítmico, teoría de control automático, estadística y análisis de series temporales para el proceso digital de señal.

La manera más fácil de visualizar Matlab es pensar en él como en una calculadora totalmente equipada. Al igual que una calculadora básica, realiza matemáticas simples como sumas, restas, multiplicaciones y divisiones. Como una calculadora científica, maneja números complejos, raíces cuadradas y potencias, logaritmos y operaciones trigonométricas tales como seno, coseno y tangente. Análogamente a una calculadora programable, puede almacenar y recuperar datos; puede crear y guardar secuencias de órdenes para automatizar el cálculo de ecuaciones importantes, puede hacer comparaciones lógicas y controlar el orden en el que se ejecutan las órdenes. De la misma forma que las calculadoras más potentes que hay disponibles, permite representar gráficamente los datos en una gran variedad de formas, ejecutar álgebra matricial, manipular polinomios, integrar funciones, manipular simbólicamente funciones, etc.

Entre sus prestaciones también se encuentran la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. El paquete Matlab dispone de dos herramientas adicionales que expanden sus prestaciones, a saber, Simulink (plataforma de simulación multidominio) y GUIDE (editor de interfaces de usuario o GUI). Además, las capacidades de Matlab pueden ampliarse con las cajas de herramientas (*toolboxes*), y las de Simulink con los paquetes de bloques (*blocksets*).

En los últimos años ha aumentado el número de prestaciones, como la de programar directamente procesadores digitales de señal (*Digital Signal Processors*, DSPs) o crear código VHDL.

2.2. Lenguaje de programación Java.

El lenguaje de programación Java, fue diseñado por la compañía *Sun Microsystems* a principios de los 90, con el propósito de crear un lenguaje que pudiera funcionar en redes computacionales heterogéneas (redes de computadoras formadas por más de un tipo de computadora, ya sean PC, MAC's, estaciones de trabajo, etc.), y que fuera independiente de la plataforma en la que se vaya a ejecutar. Esto significa que un programa de Java puede ejecutarse en cualquier máquina o plataforma. El lenguaje fue diseñado con las siguientes características:

1) **Simple:** elimina la complejidad de los lenguajes como "C" y da paso al contexto de los lenguajes modernos orientados a objetos. La filosofía de programación orientada a objetos es diferente a la programación convencional.

2) **Familiar:** como la mayoría de los programadores están acostumbrados a programar en C o en C++, la sintaxis de Java es muy similar al de estos.

3) **Robusto:** el sistema de Java maneja la memoria de la computadora por ti. No hay que preocuparse por punteros, memoria que no se esté utilizando, etc. Java gestiona la memoria del computador sin necesidad de que uno se lo indique.

4) Seguro: el sistema Java tiene ciertas políticas que evitan se puedan codificar virus con este lenguaje. Existen muchas restricciones, especialmente para los *applets*, que limitan lo que se puede y no puede hacer con los recursos críticos de una computadora.

5) Portable: como el código compilado de Java (conocido como *byte code*) es interpretado, un programa compilado de Java puede ser utilizado por cualquier computadora que tenga implementado el intérprete de Java.

6) Independiente a la arquitectura: al compilar un programa en Java, el código resultante es un tipo de código binario conocido como *byte code*. Este código es interpretado por diferentes computadoras de igual manera, solamente hay que implementar un intérprete para cada plataforma. De esa manera Java logra ser un lenguaje que no depende de una arquitectura computacional definida.

7) Multithreaded: un lenguaje que soporta múltiples *threads* es un lenguaje que puede ejecutar diferentes líneas de código al mismo tiempo.

8) Interpretado: Java corre en máquina virtual, por lo tanto es interpretado.

9) Dinámico: Java no requiere que se compilen todas las clases de un programa para que éste funcione. Si se realiza una modificación a una clase Java se encarga de realizar un *Dynamic Bynding* o un *Dynamic Loading*. Java puede funcionar como una aplicación sola o como un *applet*, que es un pequeño programa hecho en Java. Los *applets* de Java se pueden "pegar" a una página de *Web* (HTML), y así ser utilizado por cualquier usuario que disponga de un browser compatible.

2.3. Entorno de desarrollo Eclipse.

La implementación del código java de este Proyecto Fin de Máster se ha realizado con el IDE Eclipse. Eclipse es, en el fondo, únicamente un armazón sobre el que se pueden montar herramientas de desarrollo para cualquier lenguaje, mediante la implementación de los *plugins* adecuados.

La arquitectura de *plugins* de Eclipse permite, además de integrar diversos lenguajes sobre un mismo IDE, introducir otras aplicaciones accesorias que pueden resultar útiles durante el proceso de desarrollo, como herramientas UML, editores visuales de interfaces, ayuda en línea para librerías, etc.

Eclipse ha proporcionado una perfecta organización del código, localización de errores y una interfaz muy amigable de utilización, tanto para el desarrollo como para la realización de otras prestaciones como es el Javadoc.

En este documento se presenta el diseño en niveles de la API desarrollada en el transcurso del proyecto mediante la notación UML (*Unified Modeling Language*). Este lenguaje permite visualizar, especificar, construir y documentar sistemas *software*, independizando del lenguaje de programación la implementación posterior. De propósito general, es posible utilizarlo para cualquier tipo de sistema de enfoque Orientado a Objetos.

La utilización de UML requiere una herramienta de diseño CASE (*Computer Aided Software Engineering*, Ingeniería de software asistida por ordenador) para capturar los requisitos del sistema y construir el diseño del software que los resuelve.

2.4. Patrón de Diseño Objeto Activo.

El patrón **Objeto Activo** se engloba en los patrones de concurrencia. Tiene por objetivo dar orden y mejorar la concurrencia en el acceso a un objeto desacoplando la invocación de un método de este objeto de su ejecución. De esta forma se mejora la concurrencia, pues varios hilos pueden acceder al objeto intercaladamente, y la sincronización, pues sólo uno de ellos actúa sobre el objeto cada vez. A este patrón también se le conoce como objeto concurrente o en inglés *active object* o *concurrent object*.

Aunque el patrón Objeto Activo está descrito en [2], se ha incluido un resumen del mismo en esta memoria dada la importancia que tiene en el diseño del software realizado en este proyecto.

Tratar de explicar un patrón de diseño sin dar un ejemplo de uso para comprender el problema que intenta dar solución sería una labor compleja, por lo que se utilizará un ejemplo a lo largo de la explicación del patrón. Este ejemplo hace referencia a un *gateway* en un sistema de paso de mensajes. Mediante este ejemplo es bastante más sencillo comprender el patrón y el por qué tiene la estructura que tiene.

Antes de empezar, hay que aclarar que un *gateway* cumple la función de interconexión entre diferentes dispositivos o redes. Por lo tanto, el objeto activo será este *gateway*, mientras que los dispositivos o máquinas conectadas a él tendrán la función de clientes.

Ejemplo: *gateway*

Este *gateway* de comunicaciones permite la cooperación entre todos los componentes sin que haya dependencias directas entre ellos. Es un sistema distribuido, por lo que el *gateway* debe ser capaz de llevar los mensajes desde los proveedores hasta tantas máquinas destino como sean necesarias. En la figura inferior se muestra una posible estructura:

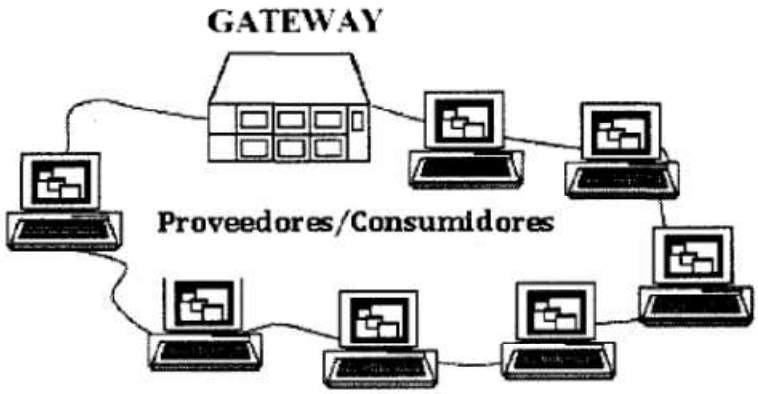


Figura 2.1. Ejemplo de gateway.

Se va a suponer que todas las partes se comunican entre sí mediante el protocolo TCP aprovechando que es un protocolo orientado a la conexión. Por tanto, el *gateway* se encontrará con los problemas de control de flujo provenientes de la capa de transporte TCP a la hora de enviar datos y deberá actuar en consecuencia.

Para más información, TCP usa el control de flujo para asegurar que fuentes de información muy rápidas o el propio *gateway* no saturan consumidores lentos o congestionan redes incapaces de almacenar y procesar los paquetes. Por ello, para mejorar la calidad de servicio para todos los componentes, el *gateway* no deberá bloquearse si el control de flujo del protocolo TCP entra en funcionamiento. Además el *gateway* debe poder escalarse eficientemente con el incremento de proveedores y consumidores.

Uno de los mejores métodos para evitar el bloqueo del *gateway* y mejorar su rendimiento es añadir concurrencia al diseño del *gateway*, por ejemplo asociar cada conexión TCP a un hilo de control diferente. Esto consigue que, aunque los hilos de las conexiones TCP afectadas por el control de flujo se encuentren bloqueados, el resto de hilos siguen funcionando con normalidad. No obstante es necesario programar los hilos de ejecución del *gateway* y cómo estos van a interactuar con los de los proveedores y los de los consumidores.

Contexto

Clientes que acceden a objetos ejecutándose en hilos de control diferentes. Esto es, máquinas que generan mensajes que han de llegar a otras máquinas pasando por el gateway.

Problema

Muchas aplicaciones intentan dar mejor calidad de servicio permitiendo que se conecten varios clientes simultáneamente. En vez de usar un único objeto pasivo que ejecuta sus métodos en el hilo de ejecución del cliente que lo invocó, un objeto concurrente tiene su propio hilo. Sin embargo, aunque los clientes se atienden concurrentemente, es necesario sincronizar los accesos a este objeto en caso de que pudiera ser modificado por varios de estos clientes a la vez.

Por tanto, el patrón intenta solucionar tres problemas:

- El acceso a este objeto por parte de los clientes no debería bloquear ni a los clientes ni al servidor de ningún modo para no degradar la calidad de servicio de todos los participantes.

- Simplificar el acceso a los objetos compartidos, de forma que toda la sincronización necesaria sea totalmente transparente para el cliente.

- Hacer un diseño donde el *software* equilibre los paralelismos entre *hardware* y *software*.

Solución
Por cada objeto afectado por los tres problemas anteriores, habrá que desacoplar la invocación de los métodos que afecten al objeto de su ejecución. La invocación tiene lugar en el hilo de ejecución del cliente, mientras que la ejecución lo hará en un hilo distinto. Además, el cliente invocará estos métodos como si fueran cualquier otro.

Entrarán en juego otros dos patrones de diseño: *proxy* y *servant*. El primero representará la interfaz del objeto activo y el segundo proporcionará su implementación. El *proxy* y el *servant* estarán en hilos de control diferentes para que la invocación y la ejecución puedan ocurrir concurrentemente. Concretamente, el *proxy* correrá en el hilo del cliente y el *servant* lo hará en otro distinto.

El *proxy* se encargará de transformar la invocación del cliente en una petición de métodos (*method request*), que será almacenada en una lista de activación (*activation list*) por un organizador (*scheduler*). El *scheduler* tendrá un hilo propio encargado de desencolar peticiones y de provocar su ejecución, por lo que el *servant* correrá en el hilo del *scheduler*.

Los clientes obtienen el resultado de la petición a través del *future* que devolvió el *proxy* durante la invocación.

Clase	**Responsabilidad**
Proxy	Define el interfaz objeto a los clientes.
Colaboradores	Crea el *Method Request* correspondiente.
Method Request	Corre en el hilo del cliente.
Scheduler	
Future	

Clase	**Responsabilidad**
Method Request	Representa la llamada al método en el objeto activo. Proporciona métodos para la sincronización cuando una petición pueda ser ejecutada
Colaboradores	
Servant	
Future	

Clase	**Responsabilidad**
Concrete Method Request	Implementa la representación de la llamada de un método. Implementa métodos de guarda
Colaboradores	
Servant	
Future	

Tabla 2.1. *Proxy*, *method request* y *concrete method request*.

El *proxy* crea un *concrete method request* durante la invocación de uno de sus métodos por parte de los clientes y lo inserta en la *activation list*. Esta lista acumula todas las peticiones realizadas y pendientes de ejecución, además de decidir qué peticiones pueden ejecutarse.

La *activation list* es la encargada de desacoplar el hilo del cliente donde reside el *proxy* del hilo donde el *servant* ejecuta la petición. El estado interno de la lista debe estar programado de tal forma que esté protegido contra accesos concurrentes que pudieran modificarla simultáneamente.

El *scheduler* corre en un hilo diferente del de los clientes, normalmente en el del propio objeto activo. Decide qué petición se ejecutará a continuación. Esta decisión se puede tomar según el criterio que se programe, por ejemplo en el orden en que fueron encoladas las peticiones o según algunas propiedades de las propias peticiones (tiempo necesario para su ejecución por ejemplo). El *scheduler* puede hacer esto gracias a los métodos de guarda del *method request* y a la información que contiene para su ejecución. El *scheduler* usa la *activation list* para organizar los *method requests* pendientes de ejecución. Los *method requests* los inserta el *proxy* cuando el cliente invoca un método.

Clase	**Responsabilidad**
Activation List	Almacena *method requests* pendientes de ejecución. El *scheduler* añade los *method request* a petición del *proxy* y los extrae cuando sea posible ejecutarlos.
Colaboradores	

Clase	**Responsabilidad**
Scheduler	Extrae *method requests* de la *activation list*. Añade *method requests* cuando el *proxy* lo solicite. Se ejecuta en el hilo del objeto activo.
Colaboradores *Activation List* *Method Request*	

Tabla 2.2. *Activation list* y *scheduler*.

Un *servant* define el comportamiento y el estado que se modela como objeto activo. Los métodos que el *servant* implementa se corresponden con aquellos a los que da interfaz el *proxy* y los *method request* que el *proxy* crea. También puede contener mecanismos que pueden usar los *method request* para sus métodos de guarda. Se invocan los métodos del *servant* cuando el *scheduler* lo solicita a través de los *method requests*, por lo que se ejecuta en el hilo del *scheduler*.

Cuando un cliente invoca un método en el *proxy*, éste le devuelve un *future*. Con este *future* el cliente es capaz de encontrar el resultado de su petición cuando esté disponible. Cada *future* reserva el espacio necesario para almacenar el resultado. El cliente puede comprobar si está disponible el resultado bien bloqueándose hasta que lo recibe o realizando sondeos.

Clase	Responsabilidad
Servant	Implementa el objeto activo
Colaboradores	Se ejecuta en el hilo del scheduler, el hilo del objeto activo.

Clase	Responsabilidad
Future	Almacena el resultado de una petición al objeto activo. Da un punto de encuentro entre el cliente y el resultado.
Colaboradores	

Tabla 2.3. *Servant* y *future*.

El diagrama de clases al que responde el patrón es el siguiente:

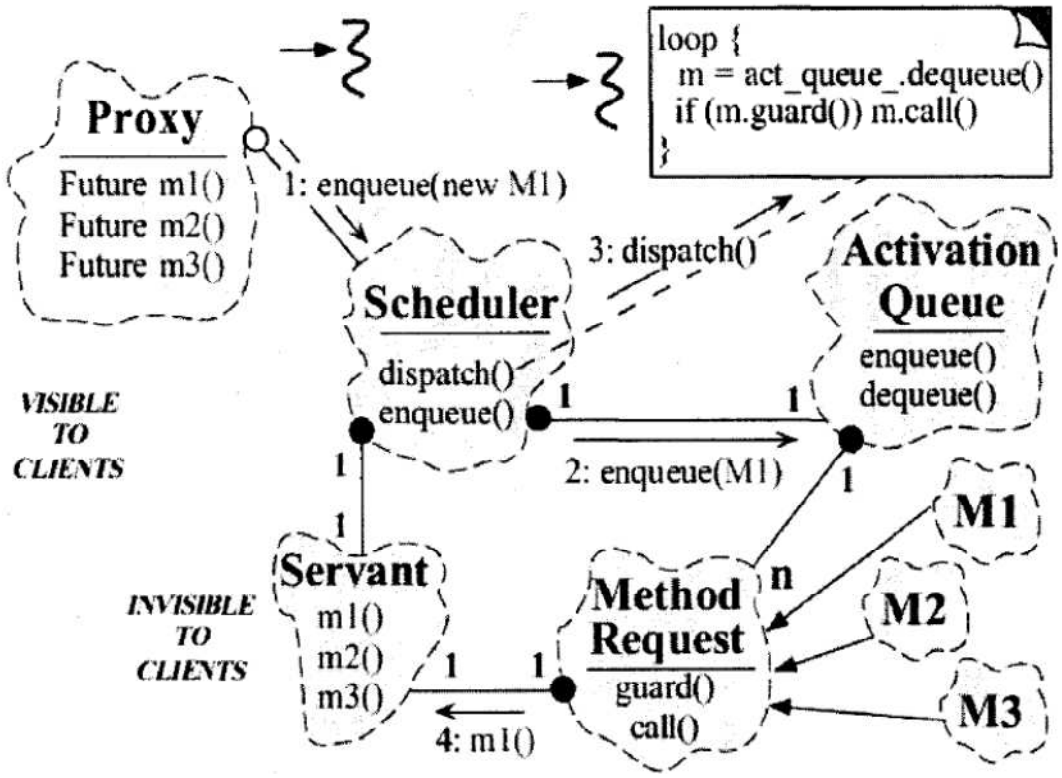


Figura 2.2. Estructura del patrón Objeto Activo.

Dinámica

El patrón se divide en tres fases:

1. Creación de un *Method Request* y su encolamiento. El cliente invoca el método en el *proxy*. Esto provoca la creación de un *method request*, que guarda lo necesario para la ejecución de la petición. El *proxy* envía al scheduler el *method request* creado para que lo encole en la *activation list*. Si la petición espera un resultado, el *proxy* le devuelve al cliente un *future* donde podrá encontrarlo cuando esté disponible. Si el método no devuelve ningún resultado, no se le devuelve al cliente ningún *future*.

2. Ejecución del *Method Request*. El *scheduler* monitoriza su *activation list* y determina qué *method request* puede ser ejecutado a continuación a través de sus métodos de guarda. Para ello tiene un hilo propio corriendo continuamente. Cuando un *method request* va a ser ejecutado, el *scheduler* lo saca de la *activation list* y se lo pasa al *servant* para que ejecute el método correspondiente. Mientras se ejecuta este método se puede acceder al estado del *servant* y crear el resultado si se requiere.

3. Finalización. En esta fase se almacena el resultado, si lo hay, en el *future* y el *scheduler* vuelve a monitorizar la *activation list* buscando *method requests* que puedan ser ejecutados. Si se esperaba un resultado, el cliente puede encontrarlo a través del *future*. Normalmente cualquier cliente que pueda encontrarse con el *future* podrá encontrarse con el resultado. Una vez el cliente se ha encontrado con el resultado, tanto el *method request* como el *future* se pueden borrar explícitamente o mediante el recolector de basura cuando ya no tengan referencias que les referencien.

Capítulo 3. Fundamentos de los Controladores Lógicos

3.1. Conceptos generales

El desarrollo de las diferentes tecnologías (mecánica, eléctrica química, etc.) a lo largo de la primera mitad del siglo XX dio lugar a una paulatina elevación de la complejidad de los sistemas e hizo que fuesen muchas las variables físicas que tuvieran que ser vigiladas y controladas. Pero dicho control no puede ser realizado de forma directa por el ser humano debido a que carece de suficiente capacidad de acción mediante sus manos y de sensibilidad y rapidez de respuesta a los estímulos que reciben sus sentidos.

Por todo ello se planteó el desarrollo de equipos capaces de procesar y memorizar variables físicas, que constituyen sistemas de tratamiento de la información. En realidad, la necesidad de estos sistemas se remonta a los primeros estados del desarrollo de la ciencia y la tecnología, pero fue el descubrimiento de la Electricidad y su posterior dominio tecnológico a través de la electrónica, el que permitió el desarrollo de sistemas que memorizan y procesan información mediante señales eléctricas con un consumo energético muy pequeño que ha permitido reducir paulatinamente su tamaño y coste. Estos sistemas que reciben el nombre genérico de "electrónicos", deben por lo tanto ser capaces de recibir información procedente de otros sistemas externos a ellos que se pueden a su vez dividir en dos grandes clases:

- Los productos industriales, que son sistemas que realizan una función determinada, como por ejemplo una lavadora, un televisor, un taladro, etc.

- Los procesos industriales, que se pueden definir como un conjunto de acciones, realizadas por una o más máquinas adecuadamente coordinadas, que dan como resultado la fabricación de un producto. Son ejemplos de procesos industriales una cadena de montaje de automóviles o una fábrica de bebidas.

Pero la mayoría de las variables físicas a medir no son eléctricas. Entre ellas se puede citar la temperatura, la presión, el nivel de un líquido o de un sólido, la fuerza, la radiación luminosa, la posición, velocidad, aceleración o desplazamiento de un objeto, etc. Por ello, el acoplamiento (*interface*) entre el sistema electrónico y el proceso productivo se debe realizar a través de dispositivos que convierten las variables no eléctricas en eléctricas y reciben el nombre de sensores (Figura 3.1).

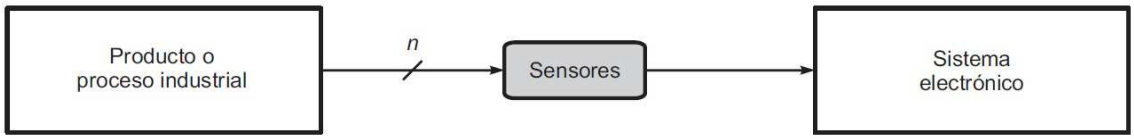


Figura 3.1. Conexión de un producto o proceso a un sistema electrónico.

Por otra parte, numerosos productos y procesos industriales generan, por medio de sensores, variables eléctricas que sólo pueden tener dos valores diferentes. Dichas variables reciben el nombre de binarias o digitales y en general se les conoce como todo-nada (*On-Off*). Los sistemas electrónicos que reciben variables binarias en sus entradas y generan a partir de ellas otras variables binarias reciben el nombre de controladores lógicos (*Logic Controllers*).

En la Figura 3.2 se representa el esquema de la conexión de un controlador lógico a un producto o proceso industrial que genera un número n de variables binarias. Dichas variables se conectan al controlador, que genera a su vez a partir de ellas m variables de salida.

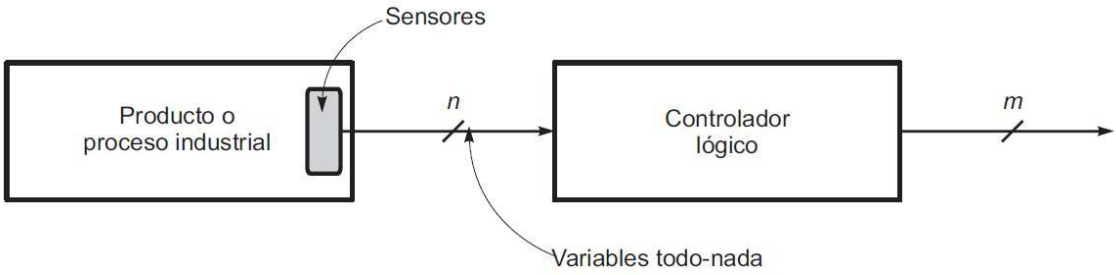


Figura 3.2. Conexión de un controlador lógico a un proceso industrial.

De acuerdo con la forma en que se utilizan las variables de salida se tiene [3]:

- Un sistema de control en bucle abierto (Figura 3.3) si las variables de salida del controlador simplemente se visualizan para dar información a un operador humano.

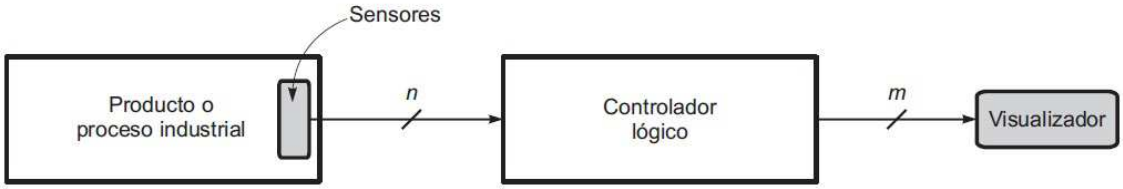


Figura 3.3. Controlador lógico en bucle abierto.

- Un sistema de control en bucle cerrado (Figura 3.4) si las variables de salida actúan sobre el producto o proceso industrial, a través de los correspondientes actuadores.

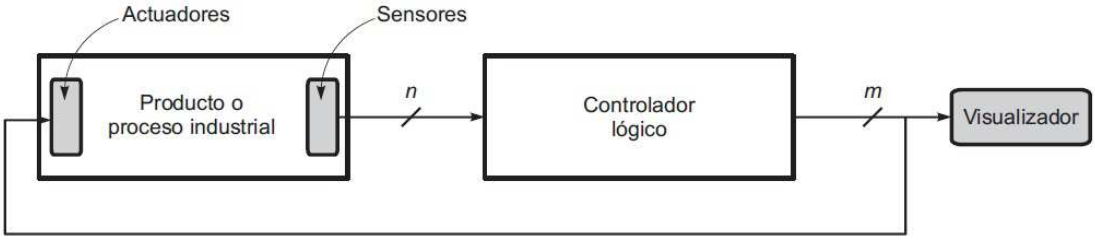


Figura 3.4. Controlador lógico en bucle cerrado.

Aunque entre los productos y los procesos industriales existe una frontera difusa, se pueden detallar las siguientes diferencias de carácter general:

- **Cambios a lo largo de la vida útil.** Los procesos suelen cambiar a lo largo de la vida útil, mientras que los productos no suelen hacerlo. Ello hace que el diseñador de los controladores lógicos utilizados en el control de procesos tenga que prever su realización de manera que se pueda modificar fácilmente el número de variables de entrada y/o salida, propiedad que recibe el nombre de modularidad. Por el contrario, al diseñar un controlador lógico para controlar con él un producto no se suele necesitar la modularidad.

- **Volumen de fabricación.** Los productos se suelen fabricar en una cierta cantidad, que depende del tipo de producto concreto. Por el contrario, los procesos industriales suelen ser ejemplares únicos que se caracterizan además por el

elevado coste de las máquinas que forman parte de ellos. Esto hace que en la selección del tipo de controlador lógico adecuado para controlar un producto se deba tener mucho más en cuenta, en general, el coste de su realización física (*Hardware*) que cuando se va a utilizar para controlar una o más máquinas que forman parte de un proceso industrial.

- **Confiabilidad.** La garantía de funcionamiento o confiabilidad (*Dependability*) se puede definir como la propiedad de un sistema que permite al usuario tener confianza en el servicio que proporciona [4]. Debido a la creciente complejidad de la tecnología, la garantía de funcionamiento de un sistema está ligada de forma creciente a diversos atributos del mismo [5] [6] [7], uno de los cuales es la seguridad, tanto en su aspecto de evitar las acciones intencionadas para dañarlo (*Security*), como para evitar que una avería del mismo o una acción anómala en el sistema controlado por él produzcan daños a su entorno o a los usuarios del mismo (*Safety*).

La complejidad de los procesos industriales, y el coste que puede tener el que una o más máquinas que lo forman quede fuera de servicio durante un tiempo elevado, hacen que la garantía de funcionamiento de los controladores lógicos utilizados en ellas sea un factor determinante en algunos campos de aplicación.

De todo lo expuesto se deduce que los controladores lógicos electrónicos deben poseer características diferentes en función de las exigencias del sistema controlado por ellos. De ahí que se puedan realizar de distintas formas que se diferencian, tal y como se indica en la Tabla 3.1, por poseer o no una unidad operativa.

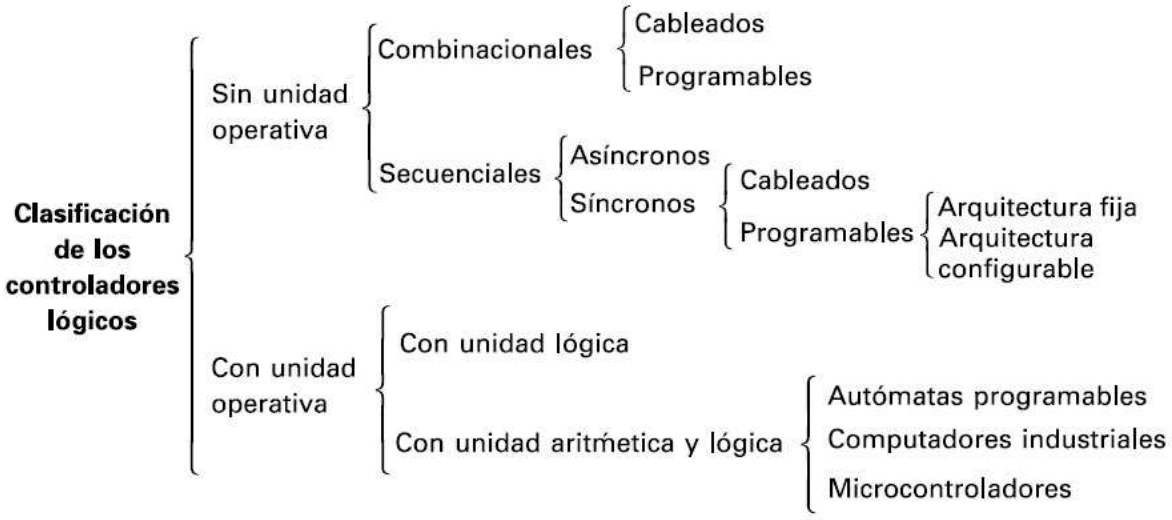


Tabla 3.1. Clasificación de los controladores lógicos.

3.2. Autómatas programables basados en un computador

3.2.1. Introducción

El progreso de la Microelectrónica permitió en la década de 1970 colocar en un solo circuito integrado la unidad central de proceso, conocida como CPU (acrónimo de *Central Processing Unit*), de un computador de una dirección bajo la denominación de microprocesador y propició el desarrollo de autómatas programables basados en un computador de arquitectura Von Neumann [8] [9]. Se comercializaron, por lo tanto, autómatas programables que poseen capacidad de proceso de variables analógicas y de ejecución de cálculos numéricos y perdió validez la definición anterior. Por otra parte, el aumento progresivo de la capacidad de los microprocesadores abarató el precio de los computadores y permitió el desarrollo de lenguajes orientados al diseño de programas de control. Todo ello ha dado lugar a una nueva definición de autómata programable como "Computador cuya organización (elementos de entrada y salida, forma constructiva, etc.) están especialmente orientadas a la implementación de sistemas electrónicos de control industrial".

Antes de analizar las características generales de los autómatas programables basados en un microprocesador es conveniente, por lo tanto, analizar las características básicas de los computadores.

3.2.2. Características generales de los computadores

Los computadores nacieron como resultado del interés por disponer de procesadores digitales programables cuando sólo existían sistemas combinacionales cableados. Se atribuye a Von Neumann la idea del procesador de programa almacenado utilizada, a partir de 1946, para desarrollar el computador EDVAC (acrónimo de *Electronic Discrete Variable Automatic Computer*) en la Universidad de Princeton. Dicho procesador poseía una unidad de control que tenía, tal como se indica en la Figura 3.5, dos estados diferenciados:

- Un estado en el cual generaba los impulsos adecuados para leer una combinación binaria situada en una memoria. Dicha combinación le indicaba a la unidad de control las señales que debía generar y por ello recibió el nombre de instrucción. A este estado se le denominaba estado de búsqueda (*Fetch*).

- Un estado en el cual generaba los impulsos adecuados para ejecutar la instrucción. Por ello se le denominaba estado de ejecución (*Execute*).

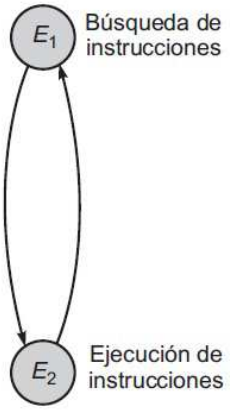


Figura 3.5. Diagrama de flujo simplificado de la UC de un computador.

Esta unidad de control debe, por lo tanto, estar conectada a una unidad de memoria de instrucciones y su interconexión con una unidad operativa formada por una unidad de memoria de datos y una unidad aritmética y lógica da lugar al diagrama de bloques de un computador representado en la Figura 3.6. La estructura de la unidad de memoria más adecuada es la de acceso aleatorio denominada RAM (acrónimo de *Random Access Memory*), que se caracteriza por que el tiempo que se tarda en leer o escribir en cualquier posición de la memoria es el mismo, independientemente de la situación de la misma [10]. Esto hace que la búsqueda de cualquier dato o instrucción utilice el mismo número de impulsos del generador, independientemente de la posición en la que esté situado.

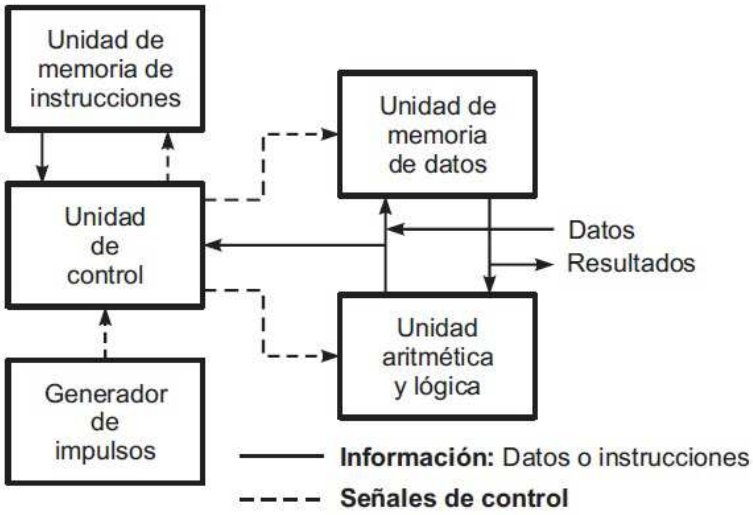


Figura 3.6. Diagrama de bloques de un computador.

El formato de la instrucción de un computador, representado en la Figura 3.7, es similar al de la instrucción de un autómata programable realizado con una unidad lógica. El campo de código de operación le indica a la unidad de control cuál es la operación que debe realizar la unidad operativa y el campo de dirección le indica la posición de la memoria de datos en la que está el operando (en el caso de que se trate de una instrucción operativa) o la posición de la memoria de instrucciones en la que está la próxima instrucción que se debe ejecutar (en el caso de una instrucción de toma de decisión o salto).

Figura 3.7. Formato de la instrucción de un computador.

En la Figura 3.9 se representa el esquema básico de la unidad de control de un computador que, para poder ejecutar un diagrama de flujo como el de la Figura 3.5, está formada por los siguientes elementos:

- Un registro en el que la unidad de control almacena la instrucción procedente de la memoria de acceso aleatorio. Recibe el nombre de registro de instrucción (*Instruction Register*).

- Un contador que contiene la dirección de la memoria de acceso aleatorio cuyo contenido se transfiere al registro de instrucción. El contenido de este contador se incrementa en una unidad cada vez que se realiza dicha transferencia y por ello

recibe el nombre de contador de programa (*Program Counter*). Además posee una entrada de información en paralelo síncrona que permite transferir el contenido del campo de dirección del registro de instrucción a su interior cuando se ejecuta una instrucción de toma de decisión o salto incondicional o condicional en la que se cumple la dirección de salto.

- Un registro de estado interno formado por un conjunto de biestables (acarreo, rebasamiento, etc.) que almacenan el valor del resultado de las operaciones realizadas por la unidad operativa. A este registro se le suele denominar palabra de estado, conocida como PSW (acrónimo de *Program Status Word*).

- Un generador de impulsos o reloj (*Clock*) y un contador que generan conjuntamente una secuencia fija de impulsos que se combinan con el campo de operación del registro de instrucción y con las salidas de los biestables del registro de estado interno para obtener las señales de control de la unidad operativa y de la memoria de instrucciones.

Pero las instrucciones, al igual que los datos, están formadas por un conjunto de variables binarias (bit) que indican a la unidad de control las acciones que debe realizar y, por ello, se pueden almacenar en la misma memoria de acceso aleatorio en la que se almacenan los datos, con lo cual se reduce el número de terminales de la unidad de control. Se obtiene de esta forma el esquema de bloques de la Figura 3.8, que es el adoptado por la mayoría de los computadores.

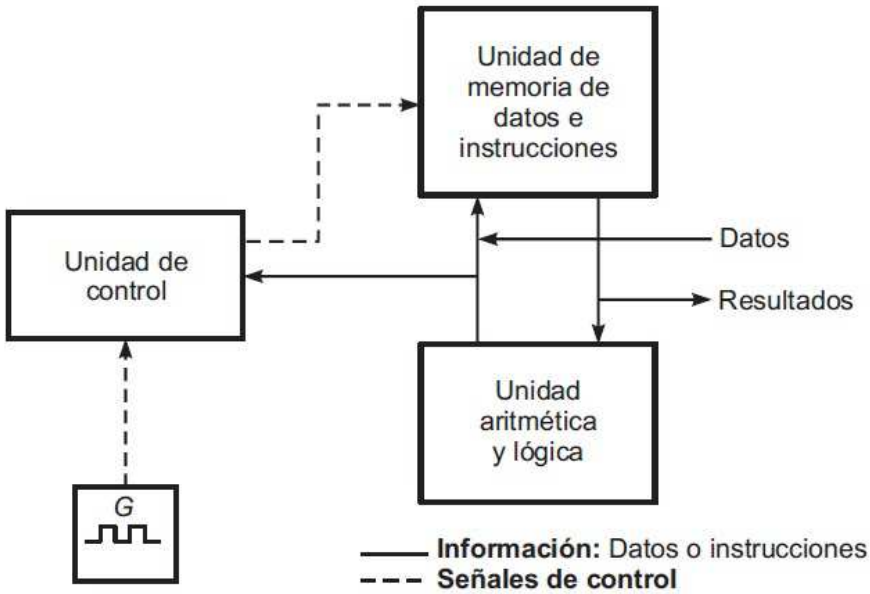


Figura 3.8. Diagrama de bloques de un computador.

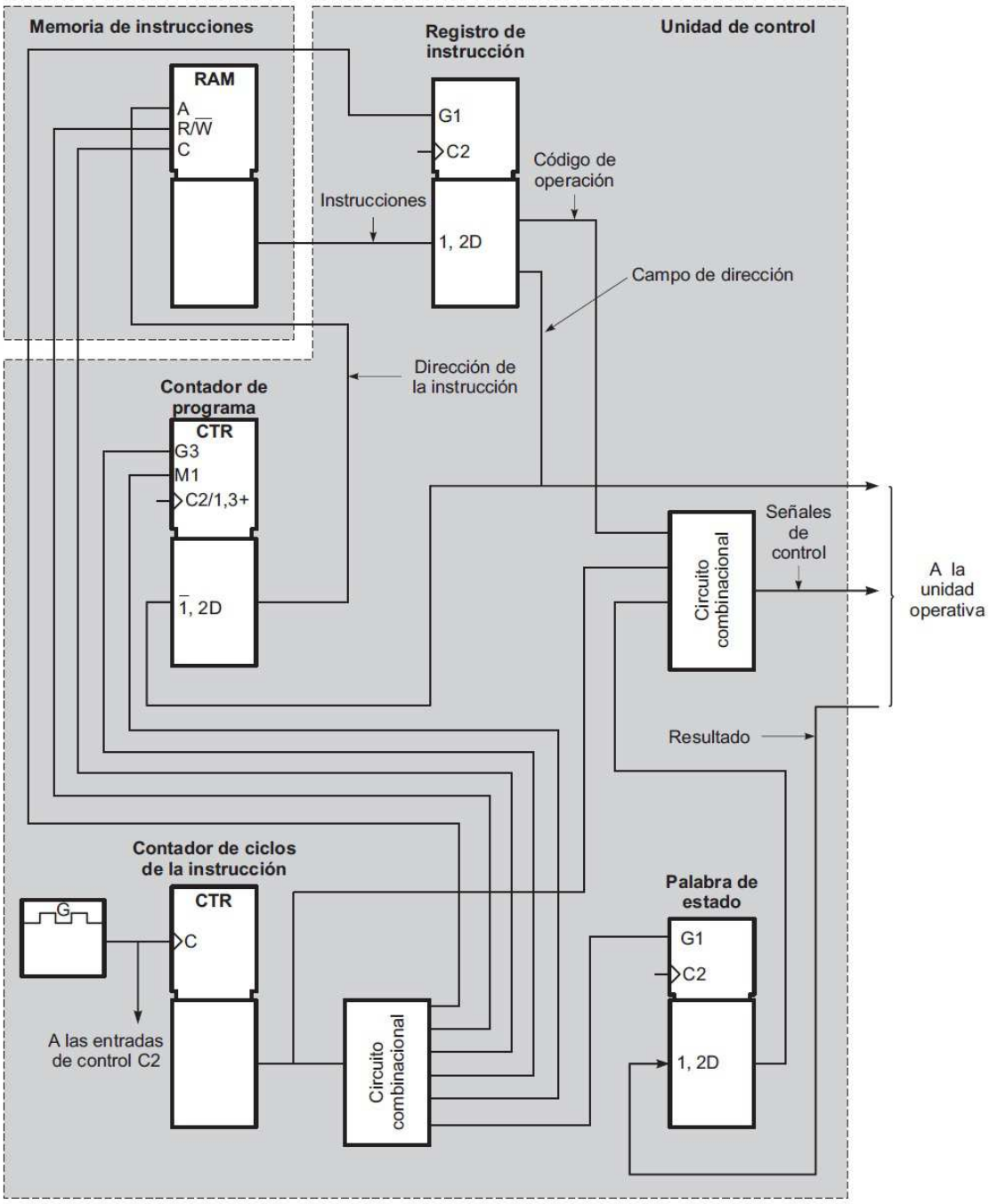


Figura 3.9. Esquema básico de la Unidad de Control de un computador.

Por otra parte, los datos externos y los resultados parciales han de memorizarse en una memoria de acceso aleatorio de escritura-lectura o activa, conocida como RAM (acrónimo de *Random Access Memory*), que se caracteriza por ser volátil, es decir, perder la información cuando se le deja de aplicar la tensión de alimentación. Las instrucciones se pueden almacenar también en una memoria activa, pero en casos en que no tengan que ser modificadas es preferible utilizar una memoria de acceso aleatorio pasiva en cualquiera de sus tipos (ROM, EPROM, EEPROM, o Flash). Las memorias pasivas de semiconductores presentan la gran ventaja de no ser volátiles y, por lo tanto, son idóneas para almacenar las instrucciones del programa.

Cuando las instrucciones se almacenan en una memoria de acceso aleatorio pasiva, el esquema de la Figura 3.8 se convierte en el de la Figura 3.10. La memoria de instrucciones y la de datos están físicamente separadas, pero se relacionan con la unidad de control a través de las mismas conexiones.

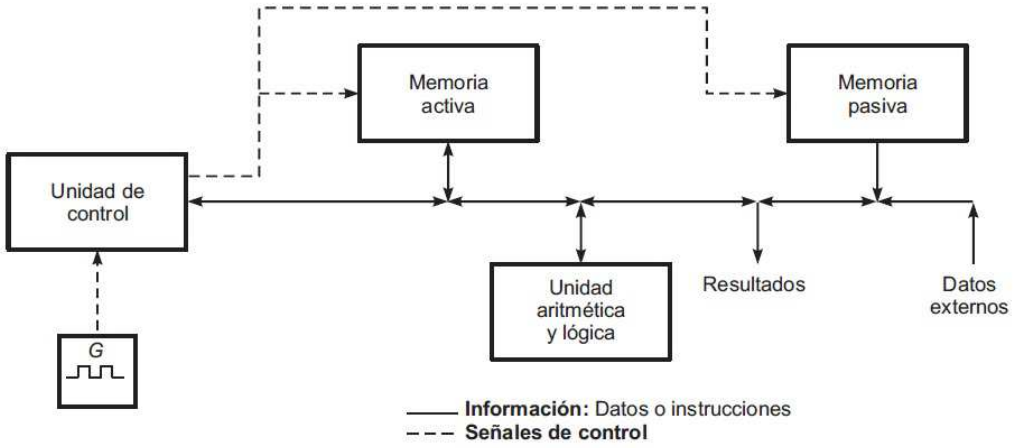


Figura 3.10. Diagrama de bloques de un computador.

De lo explicado en párrafos anteriores se deduce que la unidad de control dirige la realización del proceso y la unidad aritmética lo ejecuta. Por ello, ambos bloques se pueden reunir en uno solo que recibe el nombre de Unidad Central de Proceso (*Central Processing Unit*, CPU), tal como se indica en la Figura 3.11. El diagrama de bloques de un computador resulta, de esta manera, el indicado de la Figura 3.12. El enlace entre los dos bloques está constituido por:

- Un conjunto de señales de control que la unidad central de proceso aplica a la memoria.
- Un conjunto de conexiones por medio de las cuales la CPU envía información a la memoria o viceversa.

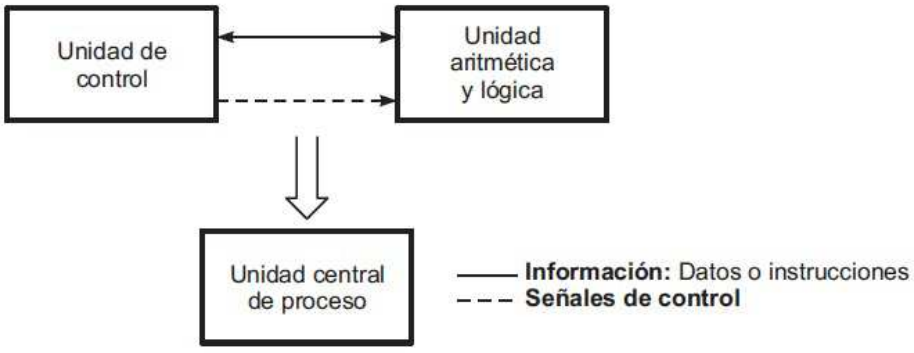


Figura 3.11. Unidad Central de Proceso (CPU).

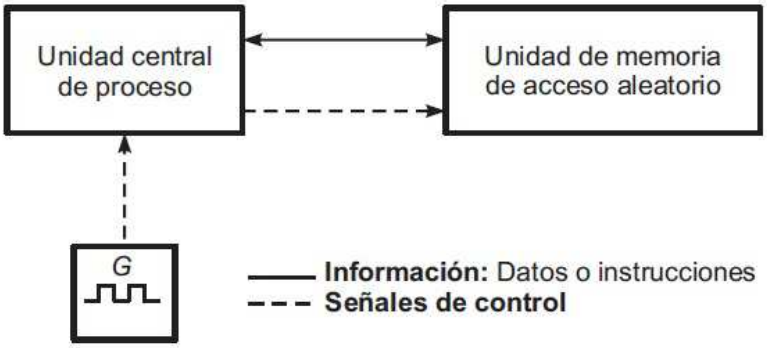


Figura 3.12. Diagrama de bloques de un computador.

En general los datos del proceso se han de transferir al computador desde el exterior y los resultados se han de transferir en sentido contrario. De igual forma, muchas veces las instrucciones se almacenan en una memoria externa y se tienen que enviar a la memoria de acceso aleatorio unida a la CPU.

Los sistemas externos se denominan periféricos y en general el computador intercambia información solamente con uno de ellos simultáneamente. Por ello la estructura típica de un computador es la representada en la Figura 3.13.

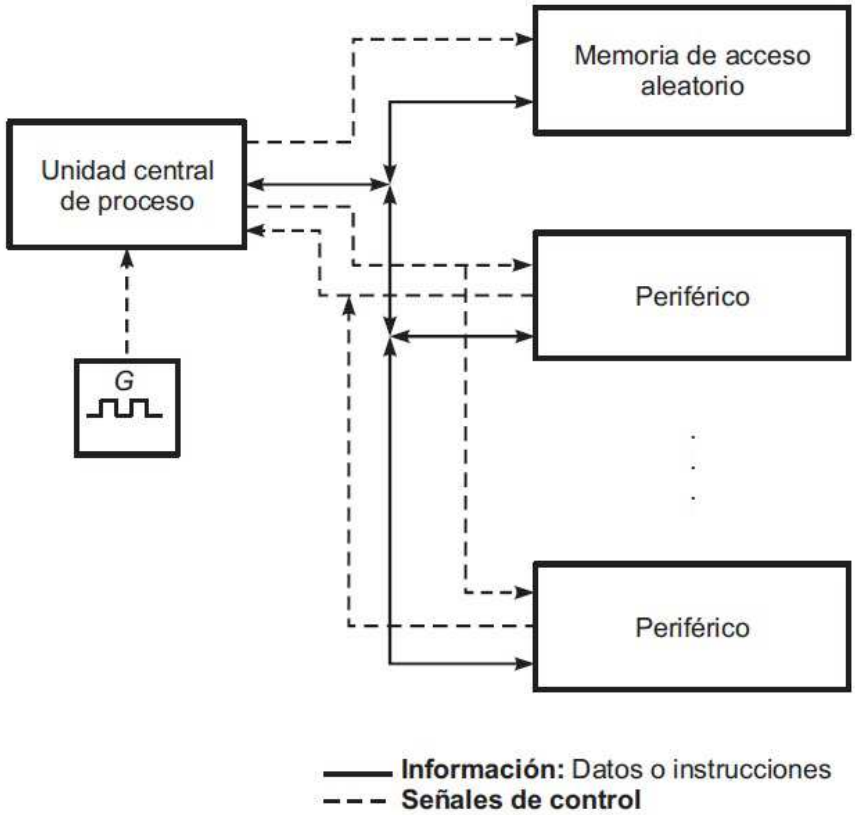


Figura 3.13. Estructura típica de un computador.

El progreso de las técnicas de integración permitió la realización de la CPU de un computador en un único circuito integrado bajo el nombre genérico de Microprocesador. El computador cuya CPU es un microprocesador se denomina microcomputador.

Por otra parte, los periféricos son en realidad sistemas digitales en la mayoría de los casos secuenciales síncronos, con un generador de impulsos distinto del que posee el microprocesador. Por ello la unión de ambos no se puede realizar de forma directa. Es necesario realizar una sincronización mediante una unidad de acoplamiento situada entre ellos que contiene una unidad de memoria cuya organización depende de las características del periférico y del programa situado en el computador, y que en los casos más sencillos es un simple registro de entrada y salida en paralelo.

Por todo lo expuesto, entre el microprocesador y cada periférico existe una unidad de acoplamiento o interfaz (*Interface*). El circuito de esta interfaz depende de las características del periférico y del tipo de proceso a ejecutar por el microcomputador. Se obtiene, de esta forma, el diagrama de bloques de un microcomputador indicado en la Figura 3.14.

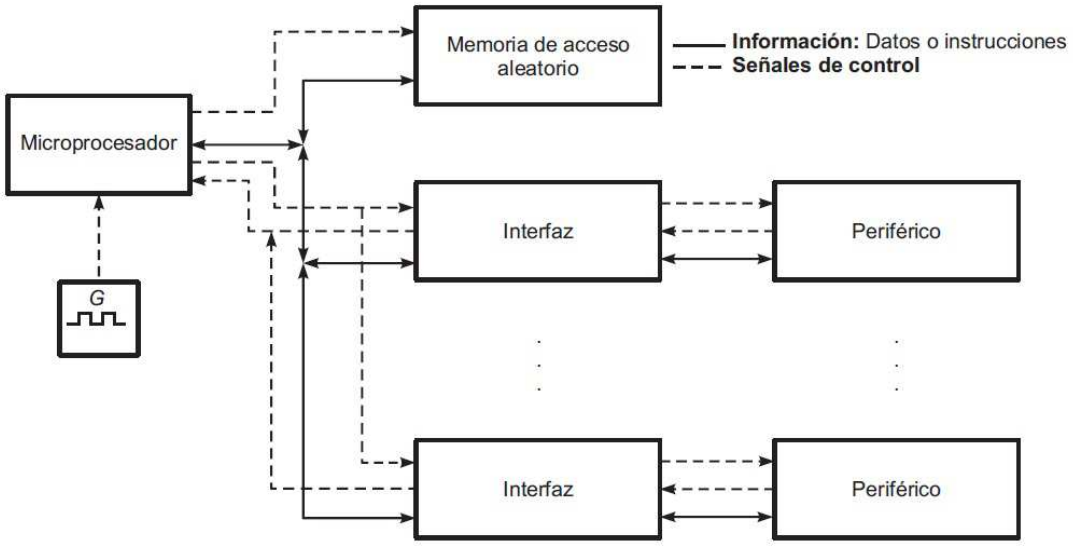


Figura 3.14. Diagrama de bloques de un microcomputador.

Además, cuando el periférico está situado a una distancia elevada del computador la transferencia de información entre ambos se hace en formato serie y la interfaz correspondiente debe generar el protocolo de comunicación entre el periférico y el computador y recibe, por ello, el nombre genérico de interfaz o procesador de comunicaciones, tal y como se indica en la Figura 3.15.

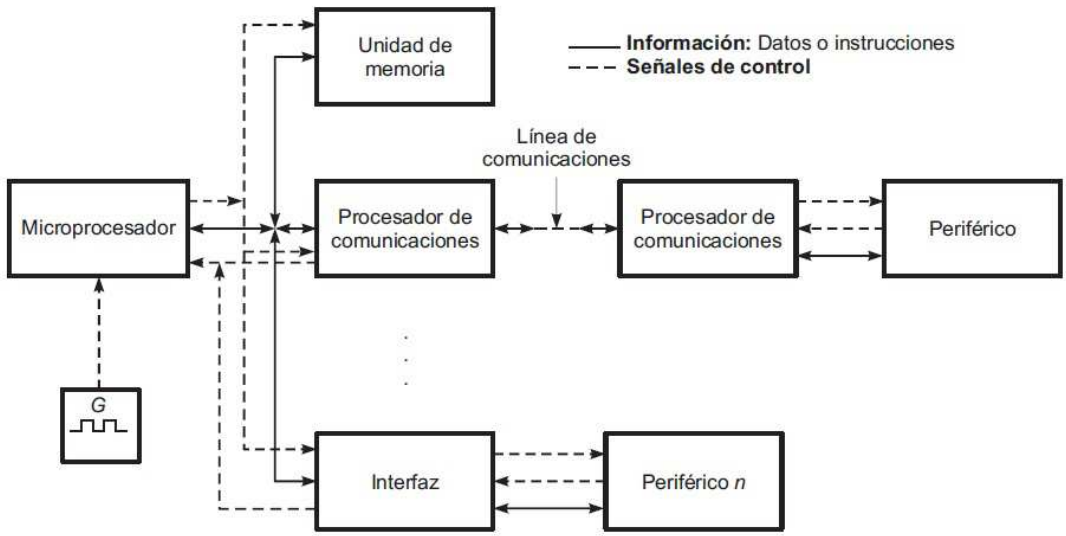


Figura 3.15. Diagrama de bloques de un microcomputador.

Si los diversos periféricos y sus interfaces se agrupan respectivamente en sendos bloques se obtiene el diagrama de bloques de la Figura XX, en el que la unidad de memoria se divide, a su vez, en dos bloques: la memoria de acceso aleatorio activa (RAM) y la memoria de acceso aleatorio pasiva (ROM, PROM o RPROM).

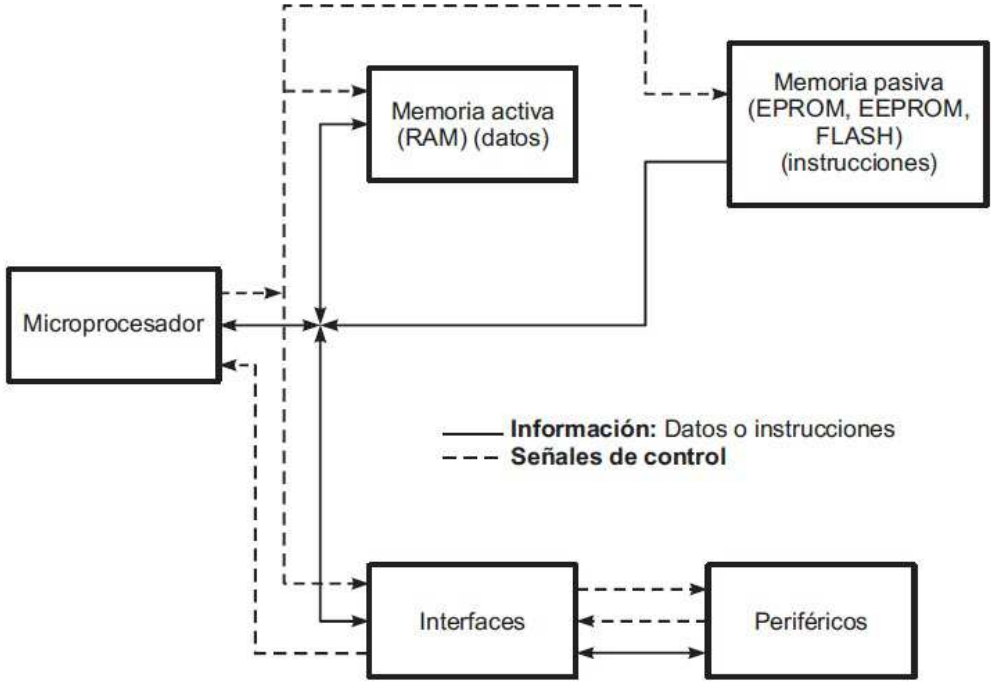


Figura 3.16. Diagrama de bloques de un microcomputador.

Las unidades de acoplamiento o interfaces son circuitos de aplicación general, cuyo diseño correcto es imprescindible para que los cambios en las especificaciones impliquen solamente una modificación de las instrucciones situadas en su memoria.

3.2.3. Características generales de los autómatas programables

Un microprocesador puede tener como unidades de acoplamiento los módulos de entrada y salida estudiados en el apartado 3.2.2.

Si en su memoria se sitúa la adecuada secuencia de instrucciones, resulta evidente que un microcomputador se puede comportar igual que un autómata programable realizado con una unidad lógica. Pero, además, un microcomputador es capaz de ejecutar un programa de control, no sólo con variables de entrada y salida digitales sino también analógicas y puede incorporar interfaces o procesadores de comunicaciones.

Se obtiene, así, el diagrama de bloques típico de un autómata programable realizado con un microprocesador representado en la Figura 3.17, que coincide con el diagrama típico de un microcomputador representado en la Figura 3.14, con las siguientes particularidades:

- Posee unidades de entrada y salida de variables digitales y analógicas así como unidades de entrada y salida especiales.

- Posee procesadores de comunicaciones para realizar su conexión con sistemas externos (unidades de desarrollo del programa de control, unidades de entrada/salida distribuida, etc.).

- Posee una unidad de memoria de acceso aleatorio dividida en tres partes que deben tener un comportamiento diferente en relación con la permanencia (volatilidad) de la información al dejar de aplicarles la tensión de alimentación.

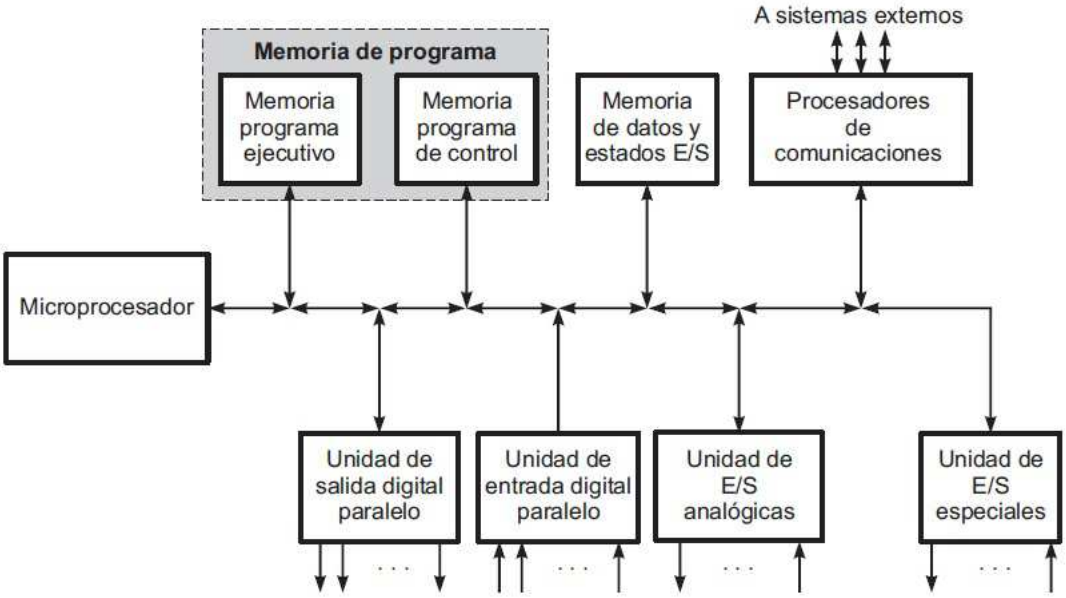


Figura 3.17. Diagrama de bloques de un autómata programable.

Antes de la comercialización de las memorias de acceso aleatorio de semiconductores se realizó con núcleos de ferrita y recibió el nombre de RAM (acrónimo de *Random Access Memories*). El desarrollo de las memorias de acceso aleatorio de semiconductores a partir de 1970 dio lugar a la comercialización de dos tipos de memorias diferentes de acuerdo a su volatilidad:

1. Las memorias de acceso aleatorio volátiles, que reciben, como herencia de las memorias de ferrita, la denominación de RAM. También se les denomina memorias de lectura y escritura (*Read/Write memories*) o activas porque el tiempo de lectura y de escritura es del mismo orden de magnitud.

2. Las memorias de acceso aleatorio no volátiles, que reciben en inglés la denominación genérica de ROM (*Read Only Memory*) porque la operación más frecuente cuando están colocadas en un sistema digital es la de lectura. En este tipo de memorias el tiempo de lectura es menor que el de escritura y, por ello, en castellano las denominamos pasivas, en contraposición con las activas. En la actualidad existen cuatro versiones de memorias pasivas:

    2.1. Las memorias de acceso aleatorio ROM propiamente dichas, cuyo contenido es establecido por el fabricante y no puede ser modificado por el usuario a lo largo de la vida útil de las mismas.

    2.2. Las memorias de acceso aleatorio denominadas EPROM (acrónimo de *Electrically Programmable Read Only Memories*). Estas memorias están realizadas con transistores MOS de puerta flotante en los que se introducen electrones mediante la aplicación de impulsos eléctricos y se extraen (borrado

de la memoria) mediante la aplicación de rayos ultravioleta. Para ello, al encapsular el circuito integrado que contiene la memoria se le dota de una ventana transparente que deje pasar los citados rayos. Si el circuito integrado carece de dicha ventana, lo cual abarata el coste de la memoria, sólo se puede programar una vez y por ello se le denomina OTP (acrónimo de *One Time Programmable*).

2.3. Las memorias de acceso aleatorio que reciben la denominación de EEPROM o E^2PROM (acrónimo de *Electrically Erasable Programmable Read Only Memories*). Estas memorias también están realizadas con transistores MOS de puerta flotante, que poseen una pequeña región túnel a través de la cual se pueden extraer los electrones de la puerta flotante mediante la aplicación de una tensión opuesta a la de programación. Las dimensiones de los transistores de efecto túnel son mayores que las de los utilizados en las memorias EPROM. Por ello las memorias E^2PROM se graban posición a posición, se usan para almacenar un número pequeño de datos y en ellas no se suele almacenar el programa de un procesador digital secuencial.

2.4. Las memorias de acceso aleatorio no volátiles, que reciben la denominación de FLASH porque la operación de grabación se realiza por bloques de posiciones en lugar de grabar posición a posición. Utilizan transistores MOS de puerta flotante en los que el espesor del aislante situado entre la misma y el canal es mucho menor que en las EPROM. Esto hace que se puedan borrar mediante impulsos eléctricos y que su densidad de integración y su velocidad sean mayores que las de las EEPROM.

El desarrollo de las memorias activas (RAM) y pasivas (ROM) ha hecho que las memorias de acceso aleatorio no volátiles, también llamadas retentivas por algunos fabricantes de autómatas programables, se puedan realizar de diferentes formas, de las que las más importantes son:

- Mediante una memoria activa (RAM) alimentada con una batería que asegura la alimentación de la memoria cuando el sistema en el que está conectada deja de recibir alimentación. Estas memorias reciben la denominación de NVRAM (acrónimo de *Non Volatil RAM*).

- Mediante una memoria E^2PROM.

- Mediante una memoria pasiva E^2PROM combinada con una activa (RAM). En el momento en el que la fuente de alimentación deja de recibir la tensión alterna de entrada, y antes de que la tensión de alimentación de la memoria RAM se anule, el contenido de la RAM se transfiere a la E^2PROM. La información se transfiere en sentido contrario cuando se vuelve a aplicar la tensión alterna de entrada. Estas memorias reciben también la denominación de NVRAM.

- Mediante una memoria pasiva FLASH.

- Mediante una memoria pasiva FLASH combinada con una activa (RAM). Su funcionamiento es similar al obtenido mediante la combinación de una memoria RAM y una E^2PROM.

A continuación se analizan las principales características del diagrama de bloques típico de un autómata programable realizado con un microprocesador (Figura 3.17).

Unidades de acoplamiento de entrada y salida de variables digitales y analógicas.

Las variables digitales se pueden acoplar a través de módulos de entrada y salida. El módulo de entrada puede estar constituido por un multiplexor realizado mediante puertas de tres estados mientras que el módulo de salida puede estar compuesto por un conjunto de biestables síncronos activados por flancos.

Por otro lado las variables analógicas se acoplan a través de unidades que realizan conversiones analógico-digitales y digital-analógicas.

Unidades de entrada/salida especiales

Son sistemas electrónicos que ejecutan un determinado proceso de información (como, por ejemplo, contaje de impulsos) de forma más eficiente que si lo ejecutara un programa.

Procesadores de comunicaciones

Permiten el enlace del autómata programable con sistemas electrónicos externos que, entre otras, pueden tener las siguientes aplicaciones:

- Unidades de programación que permiten elaborar el programa de control en diversos lenguajes y transferirlo a la memoria correspondiente del autómata programable.

- Unidades de entrada y salida de información digital y analógica remotas.

- Computadores de gestión que integran la información proporcionada por el autómata programable con otros tipos para realizar estadísticas, gráficas de producción, incidencias de funcionamiento del proceso (tiempos de parada, ritmo de fabricación, etc.).

Memoria de programa

La memoria de programa está dividida en dos partes que contienen, respectivamente:

- Un programa ejecutivo o monitor, también denominado sistema operativo (*Processor operating system*), que se encarga de realizar un conjunto de tareas imprescindibles, tales como la carga del programa de control procedente de una unidad de programa externa, el ciclo de entrada/salida de variables digitales, etc., y otros que mejoran las prestaciones del autómata programable, como por ejemplo la prueba del funcionamiento de los diferentes elementos que lo constituyen.

- El programa de control, que es diseñado por el usuario en una unidad de programación y transferido a la parte correspondiente de la memoria de programa (Figura 3.17). La modificación de este programa ha de ser posible tanto durante la fase de puesta a punto del sistema de control de la instalación como a lo largo de la vida útil de la misma y además no debe desaparecer aunque se deje de dar alimentación a la memoria que lo contiene. Por ello se suele utilizar una memoria activa (RAM) combinada con una batería o una memoria FLASH. Esta memoria actúa, así, como elemento de seguridad que permite recuperar el programa en caso de fallo de la batería.

Memoria de datos

Está constituida por una memoria de acceso aleatoria activa dividida en las siguientes partes:

- Memoria de datos o temporal del programa ejecutivo.
- Memoria de entrada y salida de variables digitales.
- Memoria de datos numéricos procedentes de convertidores analógico-digitales u obtenidos como resultados que se transmiten a convertidores digitales-analógicos. En la actualidad los módulos de entrada y salida analógicos disponen de una memoria de acceso aleatorio de doble puerto que permite que el autómata lea su contenido o escriba en ella mediante un acceso directo al periférico, según se trate de un módulo periférico de entrada o de salida, respectivamente.
- Memoria de variables internas.

La naturaleza volátil o no volátil de esta memoria depende de la aplicación. Para memorizar muchos datos se suele utilizar una NVRAM formada por una memoria RAM y una batería o una memoria FLASH

De lo anterior se deduce que todo autómata programable realizado con un microprocesador posee un mapa de memoria. En la Figura 3.18 se representa el mapa típico con todas las áreas de información que se han citado.

Figura 3.18. Mapa de memoria de un autómata programable.

3.3. Controlador de Automatización Programable CompactRIO

El controlador de automatización programable (PAC) de *National Instruments* CompactRIO es un sistema reconfigurable de control y adquisición a bajo costo diseñado para aplicaciones que requieren alto rendimiento y fiabilidad. El sistema combina una arquitectura embebida abierta con un tamaño pequeño, extrema robustez y módulos industriales de E/S intercambiables en vivo. CompactRIO es alimentado por la tecnología de arreglo de compuertas programables en campo (FPGA) de E/S reconfigurable (RIO).

3.3.1. Arquitectura CompactRIO

Los sistemas CompactRIO consisten en un chasis reconfigurable que aloja el FPGA programable por el usuario, módulos de E/S intercambiables en vivo, controlador en tiempo real para comunicación y procesamiento determinísticos y *software* gráfico LabVIEW para rápida programación de RT y FPGA. A continuación se realiza una breve descripción de los tipos de controladores así como de la gama de chasis y módulos con los que se puede configurar una arquitectura CompactRIO.

Tipos de controlador estándar.

1. Chasis y controlador integrados: *National Instruments* ofrece una línea de sistemas integrados CompactRIO que combina el alto rendimiento de la arquitectura CompactRIO con un conjunto de características menor en comparación a los sistemas modulares CompactRIO. Los sistemas integrados combinan el controlador en tiempo real CompactRIO y un plano trasero de ocho ranuras en un solo chasis que incluye el FPGA programable por el usuario. Estos sistemas integrados funcionan con el entorno de programación gráfico de NI LabVIEW fácil de usar para ayudar con el desarrollo de aplicaciones de FPGA y en tiempo real.

Figura 3.19. Chasis y controlador CompactRIO integrados.

2. Controlador de alto rendimiento: combina el controlador en tiempo real de mayor rendimiento con un plano trasero actualizable por el usuario que incluye los FPGAs más grandes y de mayor potencia de *National Instruments*. Los sistemas CompactRIO de alto rendimiento tienen la mayor cantidad de potencia de procesamiento y recursos que los convierten en ideales para aplicaciones de control avanzado, registro y transferencia de datos de alta velocidad y aplicaciones de procesamiento intenso. Los sistemas modulares funcionan con LabVIEW para ayudar con el desarrollo de aplicaciones de FPGA y en tiempo real.

Figura 3.20. Controlador de Alto rendimiento CompactRIO.

3. Controlador de rendimiento robusto: combina un controlador en tiempo real de 400 MHz con un plano trasero actualizable por el usuario que incluye los FPGAs más grandes y de mayor potencia de *National Instruments*. Los sistemas modulares también alcanzan el nivel más alto de robustez para la plataforma CompactRIO, con un amplio rango de temperatura de operación de -40 a 70 °C y rangos de impacto de 50 g; que los convierten en sistemas ideales para aplicaciones automotrices, de automatización industrial y de control avanzado.

Tipos de chasis estándar.

Además del chasis y controlador integrados descrito en los tipos de controlador estándar National Instruments ofrece la siguiente gama de chasis estándar:

1. Chasis de cuatro ranuras: chasis reconfigurable, embebido y de cuatro ranuras que forma parte de la plataforma PAC CompactRIO de alto rendimiento. Contiene un FPGA programable por el usuario Xilinx Virtex-5, lo cual le proporciona alta potencia de procesamiento y la habilidad de diseñar hardware personalizado utilizando software LabVIEW. El cRIO-9111 proporciona acceso de hardware de bajo nivel a cualquier módulo de E/S CompactRIO, y así el usuario puede crear esquemas de temporización, disparo, control y sincronización sin precedentes para aplicaciones embebidas e industriales. Todos los chasis CompactRIO incluyen orificios de montaje en panel. Existen juegos de montaje CompactRIO para realizar montajes en riel DIN o en panel.

Figura 3.21. Chasis de cuatro ranuras CompactRIO.

2. Chasis de ocho ranuras: cuando la aplicación a desarrollar requiere el más alto rendimiento y los sistemas más flexibles, la solución es la línea CompactRIO de alto rendimiento. Estos sistemas combinan el controlador en tiempo real de mayor rendimiento con un plano trasero actualizable por el usuario que incluye los FPGAs más grandes y de mayor potencia de *National Instruments*. Los sistemas CompactRIO de alto rendimiento tienen la mayor cantidad de potencia de procesamiento y recursos, lo cual hace a estos sistemas ideales para aplicaciones de control avanzado, registro y transferencia de datos de alta velocidad y aplicaciones de procesamiento intenso.

Figura 3.22. Chasis de ocho ranuras CompactRIO.

3. Chasis de expansión de ocho ranuras: el NI 9144 es un chasis robusto de ocho ranuras para módulos de la Serie C que se puede usar para añadir E/S determinística y distribuida a los sistemas PAC. Con cableado CAT 5 Ethernet estándar, se comunica de manera determinística con cualquier sistema CompactRIO o PXI en tiempo real que tenga dos puertos Ethernet. Se pueden enlazar en serie múltiples chasis NI 9144 desde el controlador para incrementar el número de canales de las aplicaciones de tiempo crítico mientras se conserva determinismo estricto con mínimos recursos del procesador.

   Este chasis está programado con el Módulo LabVIEW Real-Time para una fácil y rápida expansión del sistema en tiempo real. El NI 9144 también tiene un FPGA Xilinx Spartan de 2M de compuertas que se programa con el Módulo LabVIEW FPGA, ofreciendo una temporización de E/S personalizada y de alta velocidad, procesamiento en línea y control.

Figura 3.23. Chasis de expansión de ocho ranuras CompactRIO.

Dispositivos/Módulos

NI CompactRIO ofrece acceso directo de *hardware* a los circuitos de entrada/salida de cada módulo de E/S usando funciones elementales de E/S de LabVIEW FPGA. Cada módulo de E/S contiene acondicionamiento de señales integrado y conectores de terminal de tornillo, BNC o D-Sub. Existe una variedad de tipos de E/S disponibles, incluyendo entradas de termopares de ±80 mV, E/S analógicas de muestreo simultáneo de ±10 V, E/S digital e industrial de 24 V con capacidad de corriente de hasta 1 A, entradas digitales diferenciales/TTL con salida de suministro regulado de 5 V para codificadores y entradas digitales universales de 250 V_{rms}.

Ya que los módulos contienen acondicionamiento de señales integrado para rangos de voltaje extendidos o tipos de señales industriales, se pueden realizar sus conexiones de cableado directamente desde el módulo CompactRIO a los sensores/actuadores. En la mayoría de los casos los módulos CompactRIO permiten aislamiento del canal a tierra.

Los módulos CompactRIO conectan directamente a dispositivos RIO FPGA para crear sistemas embebidos de alto rendimiento que brindan la flexibilidad y optimización de un circuito eléctrico personalizado completamente dedicado a una aplicación de E/S. El hardware RIO FPGA ofrece opciones ilimitadas para temporización, disparo, sincronización y procesamiento de señales a nivel de sensor y toma de decisiones.

Actualmente, existen más de 50 módulos de la Serie C para diferentes medidas incluyendo de termopar, voltaje, detector de resistencia de temperatura (RTD), corriente, resistencia, tensión, digital (TTL y otros), acelerómetros y micrófonos. La cantidad de canales en los módulos individuales van desde 3 a 32 canales para alojar una amplia variedad de requerimientos del sistema.

Entre la variedad de módulos NI CompactRIO se encuentran los siguientes:

- Voltaje
- Temperatura
- Resistencia
- Tensión y puente
- Corriente
- Salidas de voltaje y corriente
- E/S Digital
- Relé
- Contador/Generación de pulso
- Acelerómetros y micrófonos
- Comunicación CAN
- Comunicación serial
- Movimiento
- Almacenamiento desmontable

3.3.2. Software CompactRIO

El software proporcionado por NI CompactRIO es LabVIEW. Con LabVIEW se pueden desarrollar aplicaciones para control industrial, adquisición de datos e interfaces hombre-máquina (HMIs) utilizando un solo entorno de desarrollo para asegurar máxima reutilización de habilidades. Las capacidades de LabVIEW y la facilidad de uso de la programación gráfica lo convierten en una excelente opción para aplicaciones que requieren:

- Medidas y análisis
- Control avanzado
- Comunicación
- HMI/SCADA

Medidas y análisis

En el caso de que se realicen medidas desde termopares, galgas extensiométricas, acelerómetros piezoeléctricos electrónicos integrados (IEPE), sensores basados en puente o codificadores de cuadratura, LabVIEW le ofrece una plataforma fiable y fácil de utilizar para recolectar datos. LabVIEW también puede realizar directamente medidas de alta precisión a millones de muestras por segundo, requeridas en aplicaciones como medidas de vibración y de calidad de potencia para monitorear el estado de una máquina. Además de los actuadores y sensores mencionados anteriormente, LabVIEW puede adquirir imágenes desde miles de cámaras y analizar las imágenes en tiempo real utilizando las librerías de imágenes de software.

Se pueden analizar los datos adquiridos utilizando algunas de las miles de funciones de análisis integradas en LabVIEW, o transferir los datos directamente al lazo de control para que se procesen posteriormente.

Control avanzado

Gracias a LabVIEW, se pueden desarrollar sistemas de control abarcando desde control proporcional integral derivativo hasta control dinámico avanzado tal como control predictivo basado en modelos. Este último ayuda al usuario a elegir el hardware y la metodología de control apropiados sin cambiar sus enfoques de desarrollo de software. Además, al utilizar la tecnología de NI SoftMotion en LabVIEW, se pueden crear controladores de movimiento personalizados utilizando LabVIEW y NI PACs para tener mejor rendimiento y flexibilidad. También se puede contar con la flexibilidad de implementar circuitos personalizados al utilizar el plano trasero FPGA en PACs CompactRIO.

Estos algoritmos de control se pueden ejecutar de manera determinista en una amplia variedad de NI PACs, tales como NI CompactRIO y NI Compact FieldPoint, que ejecutan sistemas operativos en tiempo real. Con la potencia de LabVIEW y NI PACs, usted puede ejecutar múltiples lazos PID de alta velocidad simultáneamente.

Comunicación

LabVIEW permite transferir fácilmente datos desde/hacia controladores lógicos programables (PLCs) e interfaces para operadores y hasta oficinas empresariales. Es compatible con múltiples protocolos industriales FOUNDATION fieldbus y Ethernet, tales como Modbus, OPC, EtherNet/IP, EtherCAT, CANopen, TCP/IP y serial. Por lo tanto, el usuario no está obligado a utilizar un protocolo o estándar específico para comunicarse con PLCs existentes u otros dispositivos de automatización.

Cualquiera puede establecer una conexión con su empresa y bases de datos de IT por medio de las herramientas de conectividad en LabVIEW. LabVIEW también puede colocar servicios Web en NI PACs, para conectarse a controladores de manera remota a través de un navegador de Web o cualquier aplicación de PC cliente.

HMI/SCADA

Con las herramientas basadas en configuración combinadas con las capacidades de programación de LabVIEW, el usuario puede convertir una aplicación HMI sencilla en un sistema SCADA completo con miles de etiquetas. LabVIEW incluye herramientas para registrar datos en una base de datos histórica integrada en red, encontrar tendencias de datos históricos y de tiempo real, administrar alarmas y eventos, conectar NI PACs y dispositivos OPC en un mismo sistema completo y añadir seguridad a interfaces de usuarios.

El usuario puede elegir implementar sus sistemas HMI PCs de pantalla táctil (TPCs) de NI o de terceros. También puede elegir Windows CE, Windows XP Embedded o Windows XP como sistema operativo para implementar sistemas HMI/SCADA basados en LabVIEW.

3.3.3. Características CompactRIO

Las características más importantes de NI CompactRIO para control y adquisición son:

- Control avanzado: con NI CompactRIO, se pueden desarrollar sistemas de control desde control simple de algoritmo proporcional integral y derivativo (PID) hasta control avanzado dinámico como un control predictivo de modelos. Puede ejecutar estos algoritmos de control de manera determinística y debido a la naturaleza de paralelismo del procesamiento FPGA, el añadir cálculos no reduce el rendimiento de la aplicación. Para sistemas de control de movimiento, NI SoftMotion ofrece la habilidad de crear controladores de movimiento personalizados para mejor rendimiento y flexibilidad.

Figura 3.24. Control avanzado.

- Medidas analógicas, de calidad y de alta velocidad: *National Instruments* se ha especializado, desde hace más de 35 años, en medidas de alta calidad con diseño analógico de alto rendimiento. Varias aplicaciones requieren una combinación de medidas estáticas de baja velocidad, como temperatura o medidas dinámicas de alta velocidad, como sonido y vibración. NI CompactRIO proporciona varios tipos de medidas de alta calidad, todas en un solo sistema.

Figura 3.25. Medidas analógicas de calidad.

- Procesamiento y análisis de señales: NI CompactRIO utiliza la plataforma de programación de diseño gráfico de sistemas de LabVIEW, la cual incluye miles de funciones avanzadas creadas específicamente para aplicaciones industriales de medidas y control. Se pueden usar estas herramientas para realizar fácilmente procesamiento de señales avanzado, análisis de frecuencia y procesamiento de señales digitales. Los ejemplos incluyen Transformada Rápida de Fourier (FFT), análisis de tiempo-frecuencia, sonido y vibración, análisis de onda corta, ajuste de curvas y diseño de control y simulación. También se puede extender LabVIEW con funciones específicas de la aplicación para visión artificial, control de movimiento y monitoreo de condición de máquinas.

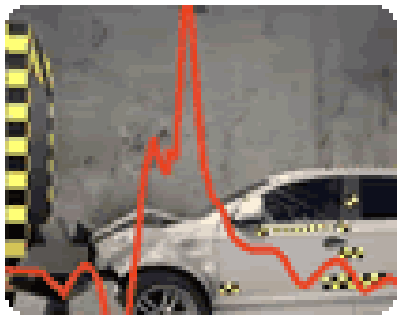


Figura 3.26. Procesamiento y análisis de señales.

- Hardware embedido, confiable y robusto: NI CompactRIO está diseñado para aplicaciones en entornos severos y lugares pequeños. El tamaño, el peso y la densidad del canal de E/S son requerimientos críticos de diseño en muchas aplicaciones embebidas. Al aprovechar el extremo rendimiento y tamaño pequeño de los dispositivos FPGA, CompactRIO es capaz de ofrecer habilidades sin precedentes de control y adquisición en un paquete compacto y robusto con certificaciones industriales y clasificaciones para operación en entornos industriales severos. Rangos de temperatura de -40° a 70° C (-40° a 158° F), rango de impacto de 50 g y una variedad de certificaciones y clasificaciones de seguridad internacional, compatibilidad electromagnética (EMC) y ambientales, están disponibles con CompactRIO.

Figura 3.27. Hardware embedido, confiable y robusto.

- Plataforma de generación de prototipos flexible y modular: NI CompactRIO tiene una variedad de controladores, chasis reconfigurable y módulos de E/S de la Serie C intercambiables en vivo para brindar la flexibilidad requerida para ir desde generación de prototipos y desarrollo al mantenerse en la misma plataforma. Actualmente, existen más de 50 módulos de E/S de la Serie C para diferentes tipos de medidas incluyendo señales de voltaje, corriente, resistencia y digitales. Los módulos específicos de sensor también están disponibles con acondicionamiento de señales integrado para sensores como termopares, RTD, tensión, acelerómetros y micrófonos. La cantidad de canales en los módulos individuales van desde 3 a 32 canales para alojar una amplia variedad de requerimientos del sistema.

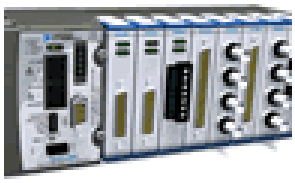


Figura 3.28. Plataforma flexible y modular.

Capítulo 4. Toolboxes Matlab

4.1. *Instrument Control Toolbox*

Matlab soporta comunicación TCP/IP gracias a la *toolbox* de Control de Instrumentos (*Instrument Control Toolbox*). Esta *toolbox* [11] ofrece funcionalidades TCP/IP para la comunicación con aplicaciones *software* remotas e instrumentos tales como osciloscopios, generadores de funciones, analizadores de señal, etc. Se trata de una colección de funciones .m desarrolladas en el entorno de Matlab que proveen las siguientes prestaciones:

- Un *framework* para comunicación con instrumentos que soportan la interfaz (IEEE-488), el estándar VISA y los protocolos TCP/IP y UDP extendiendo las prestaciones básicas de Matlab de comunicación a través del puerto serie.
- Soporte para IVI, VXI *plug&play* y drivers Matlab para instrumentos.
- Funciones para transferencia de datos entre Matlab e instrumentos con las siguientes posibilidades: los datos pueden ser binarios (numéricos) o **texto** y la transferencia puede ser síncrona con el consiguiente bloqueo de la línea de comandos de Matlab o asíncrona permitiendo el acceso a la línea de comandos de Matlab.
- Comunicación basada en eventos.
- Funciones para grabar información de datos y eventos en un fichero de texto.
- Herramientas que facilitan el control de instrumentos en un entorno gráfico de fácil utilización.
- Comunicación remota con otros ordenadores y dispositivos soportada por los protocolos TCP/IP, UDP, I2C, serial y Bluetooth.

La interfaz provista por Instrument Control toolbox ha sido empleada para la implementación del código Matlab correspondiente a las comunicaciones con la aplicación Java desarrollada.

4.2. *XML Toolbox*

La *toolbox* de XML (*XML Toolbox*) para Matlab permite a los usuarios convertir y almacenar variables y estructuras desde el *workspace* a formato de texto plano XML y viceversa. El formato XML puede ser utilizado para almacenar estructuras de parámetros, variables y resultados desde aplicaciones o bases de datos XML. Además esta *toolbox* [12] contiene rutinas de conversión bidireccionales implementadas por funciones Matlab de manejo fácil e intuitivo. Las prestaciones de XML *toolbox* son:

- Casi cualquier tipo de documento XML puede ser leído y convertido en una estructura o tipo de datos de formato Matlab.
- Las representaciones XML pueden ser almacenadas y encoladas utilizando las funciones provistas por la *toolbox Geodise Database*.
- Comparación de estructuras internas Matlab al comprobar su representación XML

- La capacidad de aprovechar las tecnologías XML y bases de datos hace que los datos estén disponibles más allá del entorno Matlab facilitando el intercambio y la reutilización por parte de los usuarios.

- El acceso a datos XML dirigido por herramientas tales como servicios Web se hace más transparente para los usuarios de ingeniería.

La interfaz provista por XML toolbox ha sido empleada para la implementación del código Matlab correspondiente a la generación de comandos XML.

Capítulo 5. Descripción del Simulador

5.1. Introducción

Uno de los objetivos de este Proyecto Fin de Máster es la implementación de un simulador en Matlab tal y como se ha mencionado en el apartado 1.2 de este documento.

En este capítulo se explica la implementación en Matlab de un simulador que modela un Panel de Control de Emergencia de un Empujador de Proa (acrónimo de *Bow Thruster*). Antes de abordar la explicación es necesario introducir una breve descripción del Empujador de Proa así como los comandos del Panel de Control de Emergencia que lo gobiernan.

Un empujador de proa o propulsor lateral de proa (Figura 5.1) es un tipo de propulsor que se utiliza en buques para aumentar su capacidad de maniobra y vuelta a puerto en caso de fallo de los sistemas propulsores primarios.

Se ha elegido este simulador por los siguientes criterios:

1. Es un caso real utilizado en la industria naval.

2. Provee un juego de instrucciones (comandos) para realizar su control.

3. Sirve como plataforma de pruebas para verificar las comunicaciones entre las aplicaciones Matlab y Java desarrolladas.

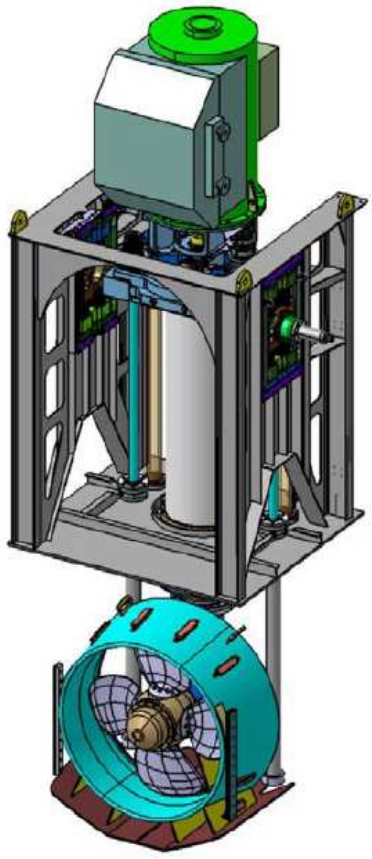


Figura 5.1. Empujador de Proa.

5.2. Operación

En este apartado se detallan los comandos que modelan las operaciones realizadas desde el Panel de Control de Emergencia del Empujador de Proa de la Figura 5.1.

5.2.1. Control de Azimuth de Emergencia (Emergency Azimuth Control)

Cuando el operador pulsa el botón 'Emergency Azimuth Control' se habilita el control de giro de azimuth desde los botones 'CW' y 'CCW'.

5.2.2. Control de giro anti horario (CCW, Counter Clock Wise rotation)

Cuando el operador pulsa el botón 'CCW' se produce el giro de azimuth en la dirección opuesta de las agujas del reloj.

5.2.3. Control de giro horario (CW, Clock Wise rotation)

Cuando el operador pulsa el botón 'CW' se produce el giro de azimuth en la dirección de las agujas del reloj.

5.2.4. Control de Empuje de Emergencia (Emergency Pitch Control)

Cuando el operador pulsa el botón 'Emergency Pitch Control' se habilita el control de empuje mediante los botones 'Left' y 'Right'.

5.2.5. Empuje hacia la izquierda (Thrust to aft direction, ←)

Cuando el operador pulsa el botón 'Left' se produce un decremento del empuje.

5.2.6. Empuje hacia la derecha (Thrust to fwd direction,→)

Cuando el operador pulsa el botón 'Right' se produce un incremento del empuje.

5.2.7. Control Manual In/Out (Manual In Out Control)

Cuando el operador pulsa el botón 'Manual In/Out' se habilita el control del botón 'Thruster Out' que permite bajar el empujador de proa (Bow Thruster).

5.2.8. Subir Empujador de Proa (Thruster In)

Cuando el operador pulsa el botón 'Thruster In' se envía la orden de subir el empujador de proa.

5.2.9. Bajar Empujador de Proa (Thruster Out)

Cuando el operador pulsa el botón 'Thruster Out' se envía la orden de bajar el empujador de proa.

5.2.10. Posición (Position In)

Cuando el operador pulsa el botón 'Position In' se verifica si se cumplen o no las condiciones para hablitar el botón 'Thruster In'.

5.2.11. Tabla resumen

A continuación se presenta una tabla resumen con los comandos descritos anteriormente:

Comando	Acción del operador	Función
Emergency Azimuth Control	Pulsación del botón 'Emergency Azimuth Control'.	Habilita el control de giro de azimuth desde los botones 'CW' y 'CCW'.
CCW	Pulsación del botón 'CCW'.	Giro de azimuth en la dirección opuesta de las agujas del reloj.
CW	Pulsación del botón 'CW'.	Giro de azimuth en la dirección de las agujas del reloj.
Emergency Pitch Control	Pulsación del botón 'Emergency Pitch Control'.	Habilita el control de empuje mediante los botones 'Left' y 'Right'.
Left	Pulsación del botón 'Left'.	Decremento del empuje.
Right	Pulsación del botón 'Right'.	Incremento del empuje.
Manual In/Out Control	Pulsación del botón 'Manual In/Out'.	Habilita el control del botón 'Thruster Out'.
Thruster In	Pulsación del botón 'Thruster In'.	Permite subir el empujador de proa.
Thruster Out	Pulsación del botón 'Thruster Out'.	Permite bajar el empujador de proa (Bow Thruster).
Position In	Pulsación del botón 'Position In'.	Verifica si se cumplen o no las condiciones para hablitar el control del botón 'Thruster In'.

Tabla 5.1. Resumen de los comandos del Panel de Emergencia

5.3. Implementación en Matlab

Para implementar el simulador en Matlab se ha utilizado el entorno GUIDE (*Graphical User Interface Development Environment*) que se incluye con Matlab a partir de la versión 5.0. De unas versiones a otras puede cambiar el aspecto de esta herramienta, aunque la teoría básica del manejo es la misma.

La creación de un proyecto mediante el entorno GUIDE se realiza de dos formas:

1. Ejecutando la siguiente instrucción en la ventana de comandos de Matlab:

>> guide

2. Haciendo clic en el icono que muestra la Figura 5.2.

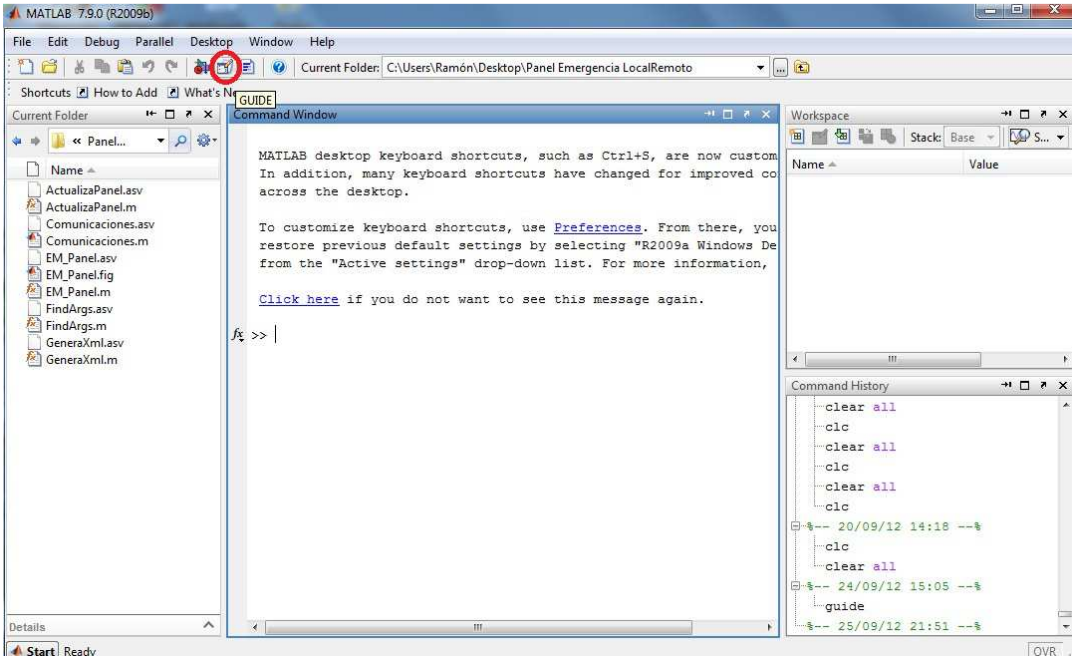


Figura 5.2. Acceso directo a entorno GUIDE de Matlab.

A continuación se presenta el cuadro de diálogo siguiente:

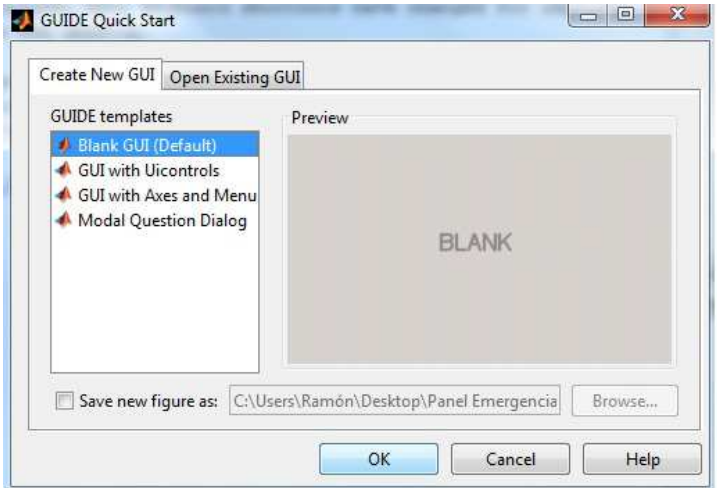


Figura 5.3. Ventana de inicio de GUIDE.

La ventana de inicio de GUIDE de la Figura 5.3 ofrece cuatro alternativas:

- *Blank GUI* (*Default*): opción de interfaz gráfica de usuario en blanco (opción por defecto), presenta un proyecto nuevo, en el cual diseñar la interfaz gráfica.

- *GUI with Uicontrols*: esta opción implementa un ejemplo en el cual se calcula la masa, dada la densidad y el volumen, en alguno de los dos sistemas de unidades. Existe la posibilidad de ejecutar este ejemplo y obtener resultados.

- *GUI with Axes and Menu*: esta opción es otro ejemplo que contiene el menú File con las opciones *Open, Print y Close*. La interfaz contiene un *Popup menu*, un *push button* y un objeto *Axes*, es posible ejecutar el programa al elegir alguna de las seis opciones que se encuentran en el menú despegable y haciendo clic en el botón de comando.

- *Modal Question Dialog*: con esta opción se muestra en la pantalla un cuadro de diálogo común, el cual consta de una pequeña imagen, una etiqueta y dos botones *Yes* y *No*, dependiendo del botón que se presione, el GUI retorna el texto seleccionado (la cadena de caracteres '*Yes*' o '*No*').

Al seleccionar la primera opción (*Blank GUI*) aparece una ventana como la mostrada en la Figura 5.4 que se denomina *Guide Control Panel*.

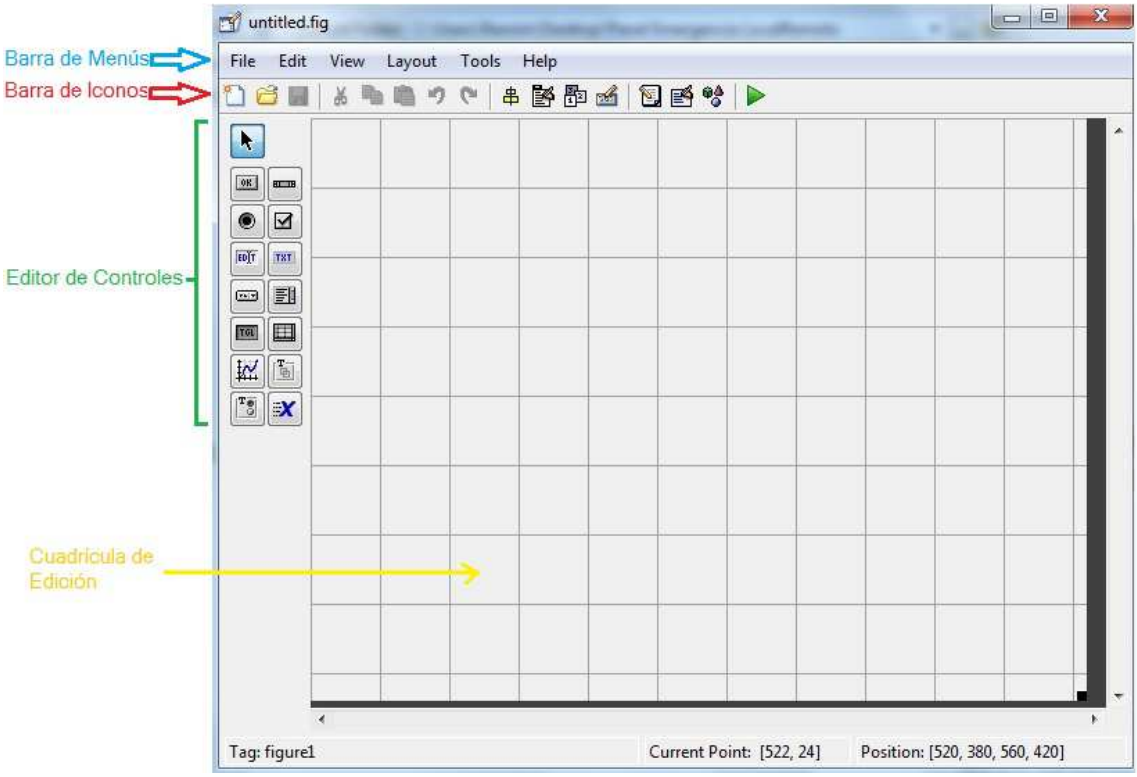


Figura 5.4. Ventana *Guide Control Panel*.

La ventana *Guide Control Panel* es un entorno con una 'Cuadrícula de Edición' en blanco sobre la que el diseñador puede dibujar la interfaz gráfica con el aspecto que desee (en el caso de estudio se va a dibujar la interfaz gráfica del Panel de Emergencia del Empujador de Arranque). Para ello el entorno dispone de varios iconos que facilitan la elección del objeto gráfico que el programador desee seleccionar

(`uicontrol` o `axes`). Estos iconos están situados en el lateral izquierdo de la ventana del entorno GUIDE, y a esta parte se le denomina 'Editor de Controles'. De esta manera se pueden ir situando los controles con el ratón y se evita la pesada tarea de introducir la posición y el tamaño de los objetos a mano. De la misma manera se pueden crear menús (`uimenu`) y editar las propiedades de los objetos de una forma fácil e interactiva haciendo clic con el ratón en las opciones de la 'Barra de Menús' y 'Barra de iconos' situadas en la parte superior del *Guide Control Panel*.

5.3.1. Editor de Controles

Como se ha comentado, el editor de controles está en la parte derecha del Guide Control Panel y sirve para seleccionar los objetos gráficos que se quieran añadir a la interfaz gráfica. Consta de varios iconos, mostrados en la xx, cada uno para cada tipo de objeto gráfico de tipo `uicontrol` o para generar unos ejes (`axis`).

El programador puede seleccionar cualquiera de estos iconos con el ratón, de manera que el puntero del puntero cambia de una flecha a una cruz, con la que se puede colocar el objeto gráfico en cualquier parte de la cuadrícula y modificar su posición y tamaño de una manera muy sencilla.

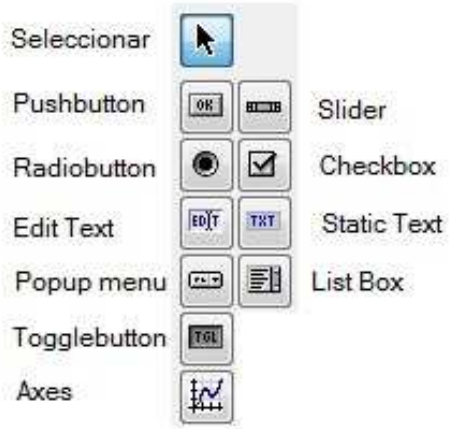


Figura 5.5. Iconos del Editor de Controles.

En el diseño de la Interfaz Gráfica de Usuario del Panel de Emergencia del Empujador de arranque se han utilizado los objetos gráficos *static text, edit text* y *pushbutton* de la Figura 5.5.

5.3.2. Editor de Propiedades (*Property Inspector*)

Las propiedades de los objetos gráficos creados se pueden consultar y modificar seleccionando un objeto y haciendo clic con el ratón sobre el icono del "Editor de Propiedades" situado en la barra de herramientas superior (Figura 5.6). También se puede acceder al Editor de Propiedades haciendo doble clic sobre un objeto situado sobre la cuadrícula. En ambos casos aparecerá una nueva ventana con una lista de las propiedades del objeto y el valor que tiene cada una, tal y como se muestra en la Figura 5.7 donde se ha creado un objeto de tipo *pushbutton*.

Figura 5.6. Icono del Editor de Propiedades.

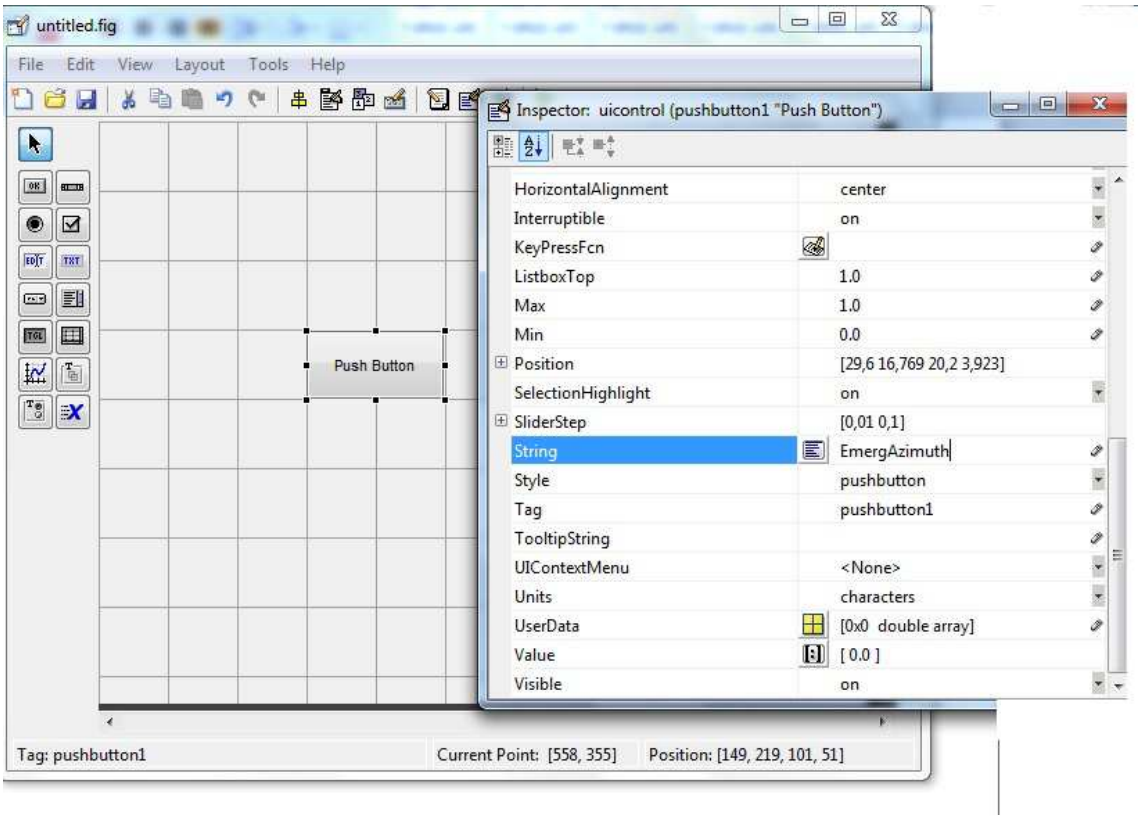


Figura 5.7. Ventana del *Property Inspector*.

Estas propiedades pueden ser modificadas utilizando el ratón y el teclado. Además, en aquellas propiedades que tengan un conjunto determinado de posibles valores a elegir aparecerá un menú con las opciones (por ejemplo en la propiedad visible del botón se puede elegir entre los valores '*on*' y '*off*'). De esta manera el programador no tiene por qué recordar los diferentes valores que pueden tomar las propiedades, facilitando la modificación de las características de los objetos gráficos. También ayuda, por ejemplo que se puedan escoger los colores de una manera más interactiva, mezclando los niveles R, G y B.

En la interfaz del Panel de Emergencia diseñada se han editado las siguientes propiedades de los objetos gráficos:

- En el caso de los *static text* se han editado los *String* que cambian el texto a mostrar y los colores del texto y del fondo (*ForegroundColour* y *BackgroundColour*).

- Para los objetos edit text se han editado las tags así como los colores del texto y del fondo (*ForegroundColour* y *BackgroundColour*).

- Finalmente en los pushbuttons se han editado los *String* que cambian el texto a mostrar, las tags de los `uicontrol`, la propiedad *Visible*, la propiedad *Enable* y los colores del texto y del fondo (*ForegroundColour* y *BackgroundColour*).

5.3.3. Diseño del Panel de Emergencia

El simulador del Panel de Emergencia del Empujador de Arranque en Matlab se puede observar en la Figura 5.8. Tal y como se puede observar en la parte superior de la ventana el simulador se ha nombrado como “EM_Panel.fig”.

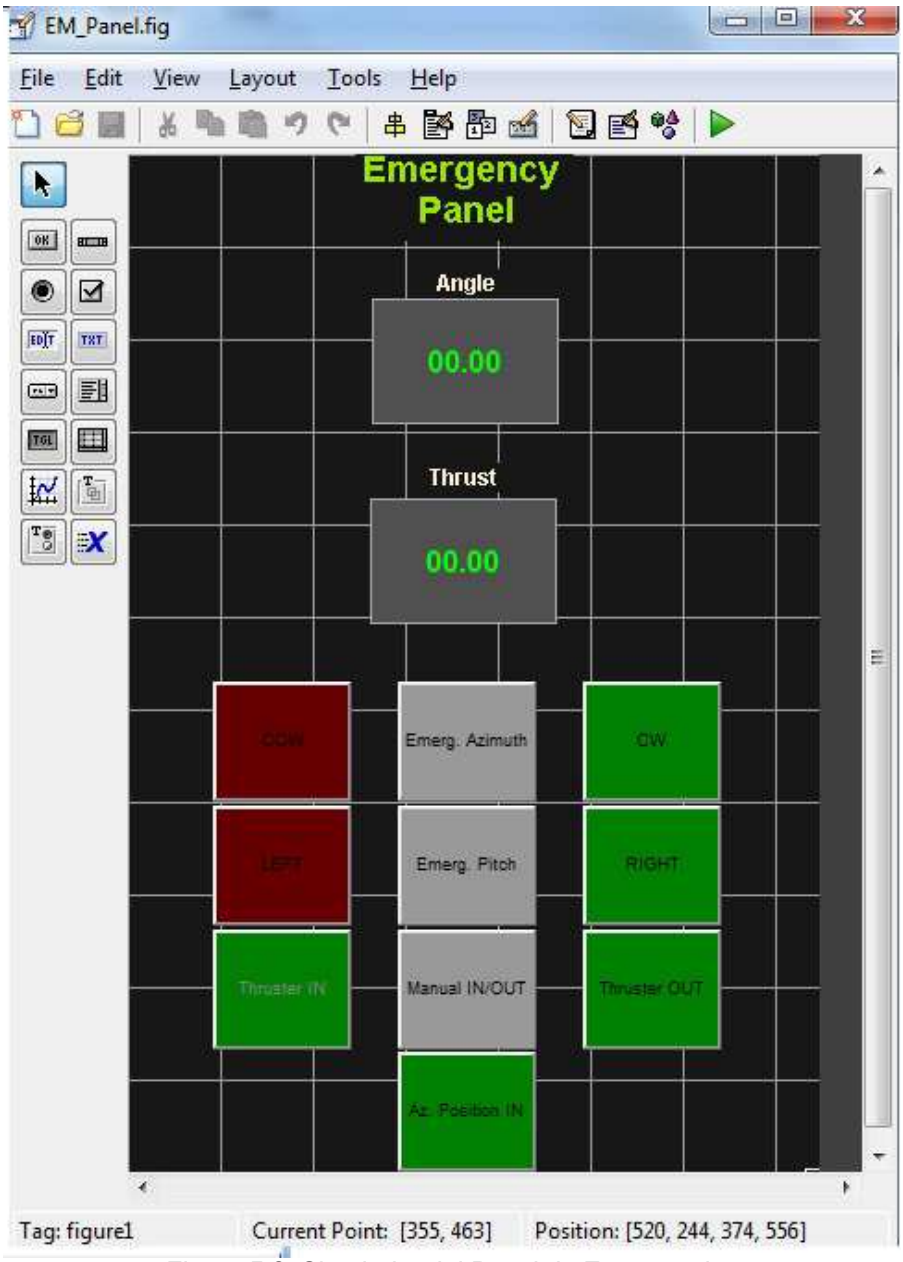


Figura 5.8. Simulador del Panel de Emergencia.

Antes de guardar los cambios hechos es importante modificar la propiedad *Tag* de cada uno de los objetos creados. Esta propiedad es una cadena de caracteres que sirve como nombre identificador de cada objeto. La edición de esta propiedad es importante por dos motivos:

1. A partir de este nombre Matlab creará automáticamente las funciones *Callback* en el fichero EM_Panel.m.

2. La estructura de *handles* que crea Matlab utilizará los *Tags* para hacer referencia a los diferentes objetos de la figura.

Una vez que se guarda la interfaz gráfica mediante la opción '*File*' → '*Save*' de la barra de herramientas superior de GUIDE automáticamente se creará otro fichero con el mismo nombre pero con extensión .m. Cuando se crea una interfaz gráfica con GUIDE, siempre habrá un fichero de aplicación .m asociado al fichero de creación de objetos gráficos .fig.

La misión del fichero .m es doble:

- Por un lado inicializa la ventana gráfica de la interfaz, creando una estructura con los handles a los diferentes objetos gráficos de la interfaz.

- Por otro lado contiene las funciones de los callback de los diferentes objetos, de manera que el programador debe modificar estas funciones para programar las respuestas a los eventos de usuario.

Finalmente se muestra la sección del código de la clase EM_Panel.fig donde se han programado las respuestas a los eventos del usuario

```
% --- Executes on button press in Pb_ThrusterOut.
function Pb_ThrusterOut_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_ThrusterOut (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
thout=get (hObject,'Value');
assignin('base','valThrusterOut', thout);

% --- Executes on button press in Pb_Left.
function Pb_Left_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_Left (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
valEmpuje=evalin('base','empuje');
valEmpuje=valEmpuje-1;
assignin('base','empuje', valEmpuje);

% --- Executes on button press in Pb_EmergAzimuth.
function Pb_EmergAzimuth_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_EmergAzimuth (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of
Pb_EmergAzimuth
val=get (hObject,'Value');
assignin('base','valAzimuth', val);

% --- Executes on button press in Pb_EmergPitch.
function Pb_EmergPitch_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_EmergPitch (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of Pb_EmergPitch
em=get (hObject,'Value');
assignin('base','valEm', em);
```

Tabla 5.2. Callbacks asociados a los eventos de usuario (I).

```
% --- Executes on button press in Pb_ManualIO.
function Pb_ManualIO_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_ManualIO (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of Pb_ManualIO
man=get (hObject,'Value');
assignin('base','valMan', man);

% --- Executes on button press in Pb_ThrusterIn.
function Pb_ThrusterIn_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_ThrusterIn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
thin=get (hObject,'Value');
assignin('base','control', thin);

% --- Executes on button press in Pb_Right.
function Pb_Right_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_Right (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
valEmpuje=evalin('base','empuje');
valEmpuje=valEmpuje+1;
assignin('base','empuje', valEmpuje);

% --- Executes on button press in Pb_CC.
function Pb_CC_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_CC (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

valAngulo=evalin('base','angulo');
valAngulo=valAngulo+1;
assignin('base','angulo', valAngulo);

% --- Executes on button press in Pb_CCW.
function Pb_CCW_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_CCW (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
valAngulo=evalin('base','angulo');
valAngulo=valAngulo-1;
assignin('base','angulo', valAngulo);

% --- Executes on button press in Pb_AzPosIn.
function Pb_AzPosIn_Callback(hObject, eventdata, handles)
% hObject    handle to Pb_AzPosIn (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
pos=get (hObject,'Value');
assignin('base','valAzPosition', pos);
```

Tabla 5.3. Callbacks asociados a los eventos de usuario (II).

Tal y como se puede observar en el código adjunto de las tablas 5.2 y 5.3, para el simulador diseñado, existen dos tipos de programación ante los eventos de usuario. El primer tipo (correspondiente a acciones de usuario sobre los botones *Emerg Azimuth, Emerg Pitch, Manual IN/OUT, Az Position In, Thruster In* y *Thruster Out*) consiste en almacenar un valor en una variable del *workspace* de Matlab cuando se produce dicho evento mientras que el segundo tipo (correspondiente a acciones del usuario sobre los botones *CCW, CW, Left* y *Right*) consiste en obtener el valor de una variable (ángulo o empuje) del *workspace* para decrementarla o icrementarla a posteriori y finalmente volver a almacenarla en el *workspace*.

Capítulo 6. Principios de diseño

En este capítulo de la memoria se explica la forma en la que se ha aplicado en el *software* el principio de diseño Objeto Activo. El código java desarrollado en el transcurso del proyecto se encuentra en la carpeta *software* que se adjunta con este documento.

6.1. Análisis del patrón de diseño Objeto Activo

En el apartado 2.4 de la memoria se ha introducido teóricamente este principio de diseño. A continuación se aborda el análisis del patrón de diseño Objeto Activo. El diseño básico del patrón cuenta con seis componentes tal y como se ha descrito en el contexto de desarrollo. Estos componentes son: ***proxy***, ***method request***, ***scheduler***, ***activation list***, ***servant*** y ***future*** y se muestran en la Figura 6.1.

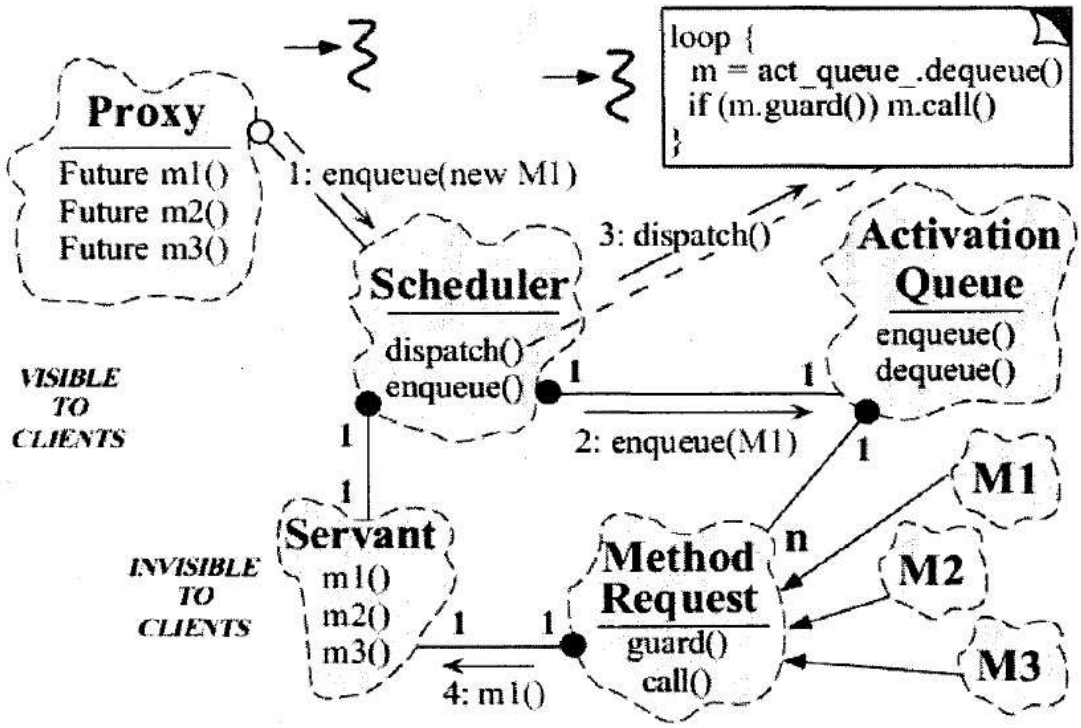


Figura 6.1. Componentes del patrón Objeto Activo.

Para empezar, comentar que, al ejecutar la aplicación, el cliente actúa en modo local. Por esta razón no es necesario ningún dispositivo o programa que realice la función de *proxy*. El diseño carece de este componente.

Cuando se ordena la ejecución de uno de los comandos estudiados en el apartado 5.2 (*Emergency Azimuth Control, Counter Clock Wise, Clock Wise, Emergency Pitch Control, Left, Right, Manual In Out Control, Thruster In, Thruster Out, Position In*) se dispara la construcción de un objeto tipo *method request.* Este objeto contiene toda la información necesaria para la ejecución del comando. El *method request* define una interfaz para la ejecución de los comandos de un objeto activo. También contiene la sincronización requerida para determinar cuándo puede ejecutarse un comando y cuando no (por ejemplo el comando *Position* habilita la subida del empujador de proa siempre y cuando se cumplan las condiciones de ángulo y empuje igual a cero). Se implementa un *method request* distinto (*concrete method request*) por cada comando

disponible. Los *concrete method request* del código del proyecto son objetos que contienen la interfaz implementada en las clases que modelan cada uno de los comandos:

EmergAzimuth
CCW
CW
EmergPitch
Left
Right
ManualInOut
ThrusterIn
ThrusterOut
Position

El *scheduler* inserta los comandos (*concrete method request*) en la *activation list*. En el código se ha modelado la activation list mediante una lista denominada `colaComandos` (ver código de la clase `HiloServidor`). Esta lista acumula todos los comandos pendientes de ejecución.

Las características fundamentales del *scheduler* en el código implementado son:

1. Corre en el hilo del propio objeto activo.
2. Se encarga de gestionar la activation list (`colaComandos`). Para ello decide que comando se ejecutará a continuación. Esta decisión depende de la posición del comando encolado (política FIFO).
3. Realiza estas funciones gracias a la información que contienen los comandos (*concrete method request*) para su ejecución.

Con lo que el *scheduler* se encuentra implícitamente en el código de la clase `HiloServidor` ya que implementa la activation list además de los métodos necesarios para su gestión:

```
public void addComando(Comando comando)
public Comando getComando()
```

Pasamos a analizar el componente *servant.* Un *servant* define el comportamiento y el estado que se modela como objeto activo. En nuestro código el *servant* implementa los métodos que se corresponden con la ejecución de los comandos. Estos métodos son los siguientes:

```
public void sendManualInOut()
public void sendThrusterOut()
public void sendEmergAzimuth()
public void sendEmergPitch()
public void sendLeft()
public void sendRight()
public void sendCw()
public void sendCcw ()
public boolean sendPosition ()
public void sendThrusterIn ()
```

Los métodos del *servant* se invocan cuando el *scheduler* lo solicita a través de los *method request* por lo que se ejecuta en el hilo del *scheduler*. El *servant* se encuentra implícitamente en el código de la clase `HiloServidor` por dos razones:

1. Implementa su interfaz.
2. Implementa el hilo del *scheduler*.

Para finalizar con el análisis del patrón Objeto Activo mencionar que en el diseño no existe la implementación del *future* ya que no hay que guardar ningún resultado al ejecutar cada uno de los comandos del panel de emergencia.

Capítulo 7. Paquetes y librerías

7.1. Diseño en niveles de la API

En este apartado se describe el diseño de la API. A continuación se muestra el diagrama de relaciones de uso de los diferentes paquetes que componen el proyecto desarrollado en Eclipse.

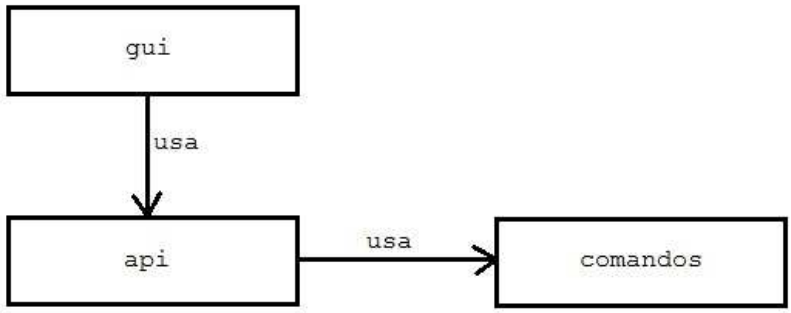


Figura 7.1. Diseño de la API.

Tal y como muestra la Figura 7.1 se ha realizado un diseño de 2 niveles. En el nivel inferior se encuentra el paquete `comandos` que almacena las clases que implementan los comandos del Panel de Emergencia del *Bow Thruster* en java. El segundo paquete, llamado `api`, es el más importante de todo el diseño ya que contiene la clase `HiloServidor`. En el capítulo anterior se ha explicado cómo se ha aplicado el principio de diseño "Objeto Activo" en la implementación de esta clase.

Por último en el nivel más alto se encuentra el paquete `gui` que tal y como indica su nombre contiene la interfaz gráfica de usuario implementada en Java que se comunica con el simulador Matlab descrito en el apartado 5.3.

7.2. Paquete de comunicaciones

Para implementar las comunicaciones TCP/IP en Java se ha empleado el paquete `java.net` que proporciona una clase *Socket*, la cual modela una de las partes de la comunicación bidireccional entre las aplicaciones Java y Matlab desarrolladas.

La clase *Socket* se sitúa en la parte más alta de una implementación dependiente de la plataforma, ocultando los detalles de cualquier sistema particular al programa Java. Usando la clase `java.net.Socket` en lugar de utilizar código nativo de la plataforma, la aplicación Java implementada puede comunicarse a través de la red de una forma totalmente independiente de la plataforma.

De forma adicional, `java.net` incluye la clase `ServerSocket`, la cual implementa un *socket* que el servidor puede utilizar para escuchar y aceptar peticiones de conexión del cliente (simulador Matlab).

7.3. Librería de tratamiento XML

En el código del proyecto Java implementado se ha utilizado la versión 1.1.3 de la librería JDOM [13] que se adjunta en la carpeta *software* anexa a la memoria del Proyecto Fin de Máster. JDOM es una API desarrollada específicamente para Java que da soporte al tratamiento de XML: parseo, búsquedas, modificación, generación y serialización. Se trata de un modelo alternativo a la librería DOM que fue desarrollada para que fuera independiente del lenguaje mientras que JDOM está creado y optimizado específicamente para Java. Esto imprime a JDOM las ventajas inherentes a Java, que lo convierten en una API más eficiente y natural de usar para el desarrollador Java y por tanto requiere un menor coste de aprendizaje.

De la misma forma que DOM, JDOM genera un árbol de nodos al parsear un documento XML. En cuanto a lo tipos de nodos, son similares a los de DOM, aunque algunos cambian el nombre ligeramente. La jerarquía de clases también se ve modificada.

Mientras que en DOM todas las clases heredan de la clase Node, en JDOM heredan de la clase Content, que se encuentran en el paquete `org.jdom`, con la excepción de las clases `Document` y `Attribute` (`Attr` en DOM). La razón es que `Document` no se considera un contenido, sino más bien el continente y `Attribute` tampoco se considera un nodo, sino una propiedad de los nodos tipo `Element`.

Finalmente otra diferencia entre DOM y JDOM es que mientras en DOM todos estos tipos de nodos eran interfaces, en JDOM son clases y, por tanto, para crear un objeto de cualquier tipo basta con usar el operador *new*.

Capítulo 8. Desarrollo de implementación

En este apartado de la memoria vamos a citar los fragmentos de código que requieren de un análisis más detallado debido a la dificultad de su programación como es el caso de las comunicaciones entre las aplicaciones Java y Matlab desarrolladas, o bien, debido a su importancia para el correcto funcionamiento como es el caso de la generación/recepción de comandos Xml en ambas aplicaciones.

Para que el resto del *software* desarrollado sea perfectamente legible y fácilmente modificable se ha comentado adecuadamente gracias a la herramienta *Javadoc* que proporciona el entorno de desarrollo Eclipse.

8.1. Comunicaciones Java/Matlab

Para implementar las comunicaciones entre las aplicaciones Java y Matlab se han utilizado las funciones del paquete java.net (implementación en Java) y de la *Instrument Control Toolbox* (implementación en Matlab) tal y como se ha mencionado en los apartados 7.2 y 4.1 respectivamente.

En el código de la clase Java `PanelRemotoEmergencia` se han instanciado las clases `Socket` y `ServerSocket`. El servidor establece un puerto y espera durante un cierto tiempo (*timeout* segundos), a que el cliente establezca la conexión. Cuando el cliente solicita una conexión, el servidor la abrirá con el método `accept()`. El cliente establece la conexión con la máquina host a través del puerto 37500.

```
//Abrir conexion
serverSocket = null;

try {
	System.out.println("Conexion abierta.................");
	serverSocket = new ServerSocket(37500);

} catch (IOException e) {
	System.err.println("No se puede escuchar en el puerto: 37500");
	System.exit(1);
	}

clientSocket = null;

try {
	System.out.println("Esperando.................");
	clientSocket = serverSocket.accept();

} catch (IOException e) {
	System.err.println("Fallo en la conexion");
	System.exit(1);
	}

out = new PrintWriter(clientSocket.getOutputStream(), true);
in=new
BufferedReader(newInputStreamReader(clientSocket.getInputStream());
```

Tabla 8.1. Implementación de comunicaciones en Java.

Tal y como se puede observar en el código expuesto en la

Tabla 8.1 la comunicación se realiza gracias a las clases `PrintWriter` y `BufferedReader` que permiten escribir y leer caracteres en/de el flujo de salida/entrada respectivamente.

En el código de la función Matlab `Comunicaciones` (Tabla 8.2) se ha creado un objeto TCP/IP que establece la conexión a través del puerto 37500. Las funciones `fopen`, `fread` y `fwrite` de *Instrument Control Toolbox* implementan el establecimiento de la conexión así como la lectura y escritura de comandos enviados/recibidos a/desde Java.

```
%Objeto TCP/IP
t = tcpip('localhost',37500);
fopen(t);

%Lectura del comando enviado por Java
a = fread(t, 127);
comando_java = char(a');
flushinput(t);

%Envio de comando a Java
fwrite(t,comando_matlab);
flushoutput(t);
```

Tabla 8.2. Implementación de comunicaciones en Matlab.

8.2. Tratamiento XML Java/Matlab

En la clase Java `HiloServidor` se realiza la invocación del método `generaXml()` para construir los comandos que se envían a la aplicación Matlab (Tabla 8.3).

```
    public String generaXml (){

      //elemento root
        Element nam = new Element(source);
        //hijo del root
        Element comando=new Element(name);
        nam.addContent(comando);
        Element id = new Element("Id").setText("-"+Id+"-");
        nam.addContent(id);
        Element param = new Element("value").setText("-"+value+"-");
        nam.addContent(param);

        Document doc = new Document(nam);//Creacion del documento

        Format format = Format.getPrettyFormat();

        // Serializador con el formato deseado
        XMLOutputter xmloutputter = new XMLOutputter(format);

        // Serializar el document parseado
        String docStr = xmloutputter.outputString(doc);

      return docStr;

    }
```

Tabla 8.3. Generación Comandos Xml en Java.

Para la generación de cada comando Xml Java se emplea las funciones de la librería JDOM descrita en el apartado 7.3. así como los campos que definen a cada comando `source, name, id y value` según se indica en la Tabla 8.4.

source	name	id	value
ComandoJava	ManualInOut	10	0/1
ComandoJava	ThrusterOut	20	0/1
ComandoJava	EmergAzimuth	30	0/1
ComandoJava	CCW	31	0/1
ComandoJava	CW	32	0/1
ComandoJava	EmergPitch	40	0/1
ComandoJava	Left	41	0/1
ComandoJava	Right	42	0/1
ComandoJava	Position	50	0/1
ComandoJava	ThrusterIn	60	0/1

Tabla 8.4. Campos de comandos Xml Java.

La Tabla 8.5 ilustra la estructura del comando 'ThrusterOut':

```
<?xml version="1.0" encoding="UTF-8"?>
<ComandoJava>
  <ThrusterOut />
  <Id>-20-</Id>
  <value>-0-</value>
</ComandoJava>
```

Tabla 8.5. Comando ThrusterOut.

En la función de Matlab `Comunicaciones` se realiza la invocación del método `GeneraXml` que construye los comandos Matlab con *XML toolbox* descrita en el apartado 4.2. así como los campos que definen a cada comando `source` que en este caso es ComandoMatlab, `name, id y value` según se ha indicado en la Tabla 8.4.

```
%Comando de control en xml
project.name = 'ComandoMatlab';
project.v_angulo = v_ang;
project.v_empuje=v_thr;
project.v_emergAzimuth=v_az;
project.v_emergPitch=v_Em;
project.v_ManualIO=v_Man;
project.v_thrO=v_thrOut;
project.v_PositionInOut=v_pos;
project.v_thrustIn=v_thrIn;

comando_matlab=xml_format(project,'off');
```

Tabla 8.6. Generación de Comandos Xml en Matlab.

Para interpretar correctamente las cadenas de caracteres (comandos) en recepción se han provisto de delimitadores en los comandos (carácter delimitador **?** para comandos Matlab y carácter delimitador **-** para comandos Java). En el código java se ha utilizado la clase `StringTokenizer` para encontrar los campos de cada comando mientras que en Matlab se ha implementado un bucle *for* que recorre la cadena de caracteres recibida identificando las posiciones en las que se encuentran los delimitadores y, por tanto, los campos de los comandos.

Capítulo 9. Manual de Usuario

Este capítulo sirve como manual de las Interfaces Gráficas de Usuario (GUIs) desarrollado en este proyecto.

En primer lugar arrancar los entornos Eclipse y Matlab. A continuación abrir los proyectos desarrollados en ambos entornos. Realizar los siguientes pasos:

1. Ejecutar la clase Java `PanelRemotoEmergencia` en Eclipse con lo que aparecen los mensajes “Conexión Abierta” y “Esperando” en la consola tal y como ilustra la Figura 9.1.

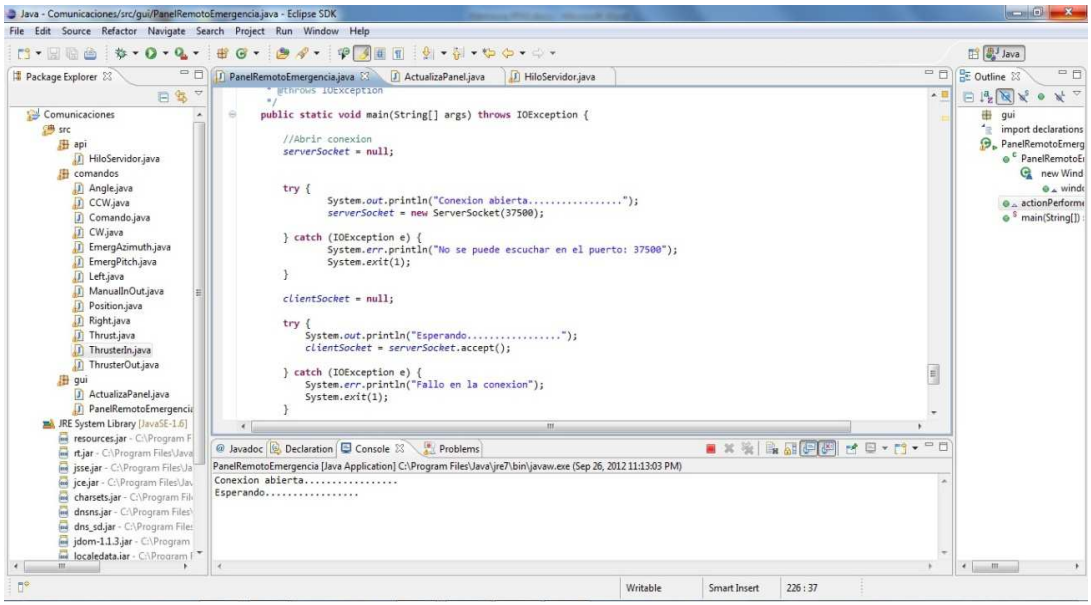


Figura 9.1. Ejecución de la clase `PanelRemotoEmergencia`.

2. Al ejecutar la clase `Comunicaciones.m` en Matlab se muestran en pantallas las Interfaces Gráficas de Usuario de las aplicaciones desarrolladas (Figura 9.2).

Figura 9.2. Ejecución de la clase `Comunicaciones.m`

3. Hacer clic con el ratón en el botón 'Manual IN/Out' de una de las interfaces. Posteriomente aparecen en ambas ventanas los botones 'Thruster OUT' (habilitado) y 'Thruster IN' (deshabilitado) tal y como muestra la Figura 9.3.

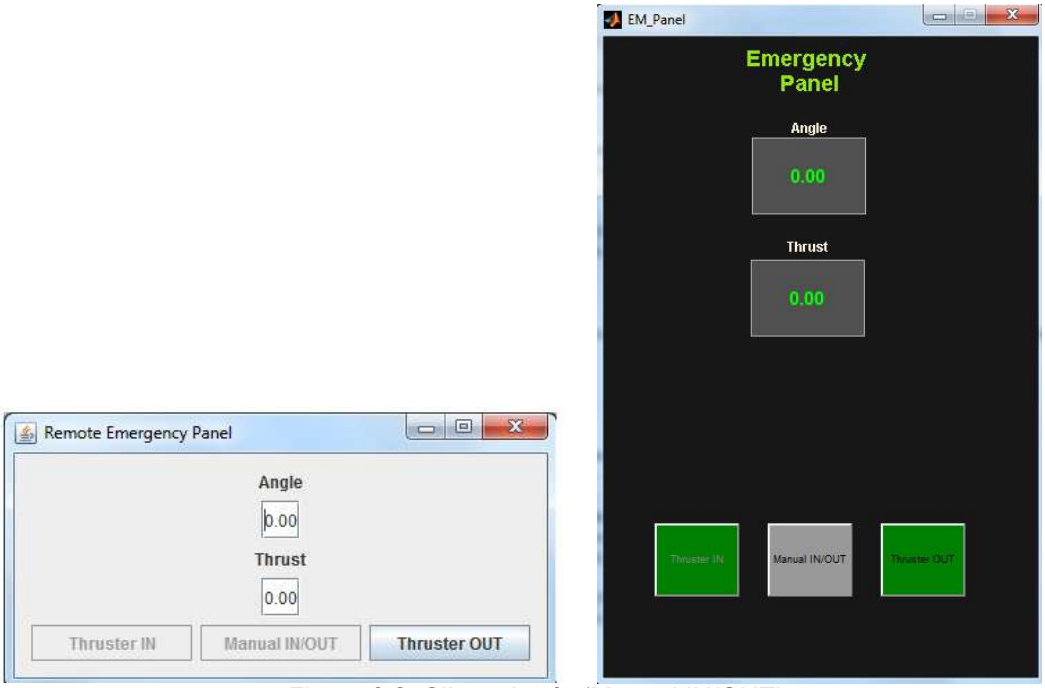


Figura 9.3. Clic en botón 'Manual IN/OUT'.

4. Hacer clic con el ratón en el botón 'Thruster OUT' de una de las ventanas. En la Figura 9.4 se observa que aparecen los botones 'Emerg. Azimuth' , 'Emerg Pitch' y 'Az. Position IN' (habilitados). Aunque se muestren los botones 'Thruster IN', 'Thruster Out' y 'Manual IN/OUT', se encuentran deshabilitados. Ambas ventanas muestran la misma información.

Figura 9.4. Clic en botón 'ThrusterOut'.

5. Hacer clic con el ratón en el botón ‘Emerg Azimuth’ y posteriormente en el botón ‘Emerg. Pitch’ de cualquier ventana. Tal y como se puede observar en la Figura 9.5 los únicos botones en este estado de la ejecución que se encuentran habilitados son ‘CCW’, ‘CW’, ‘Left’, ‘Right’ y Az.Position.

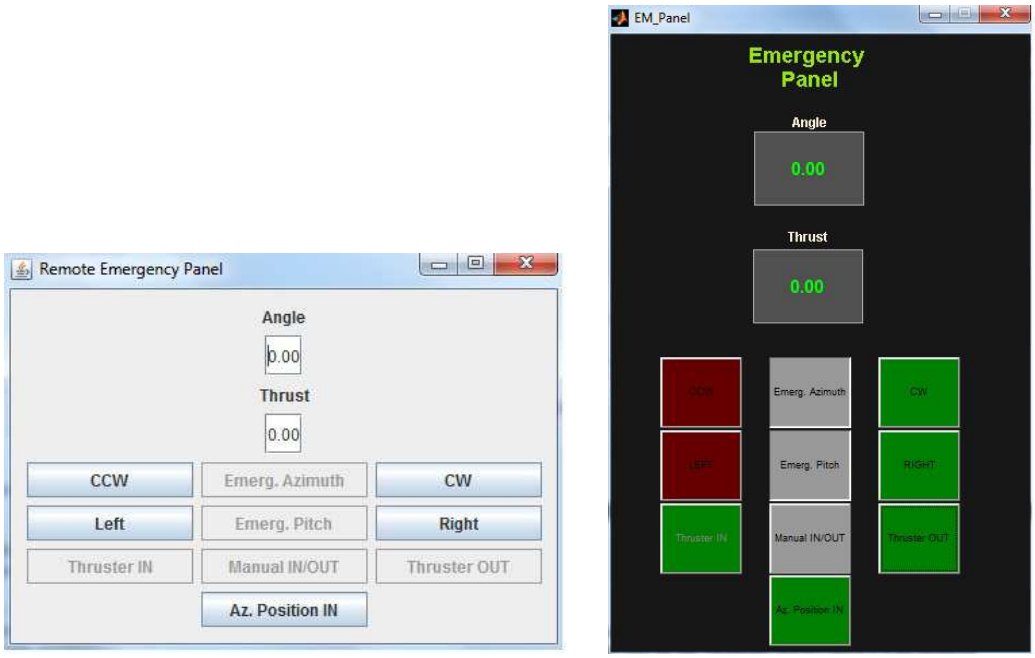


Figura 9.5. Clic en botones ‘Emerg Azimuth’ y ‘Emerg Pitch’.

6. Hacer clic en los botones ‘CCW’ y ‘CW’ para variar el ángulo en sentido anti horario y horario respectivamente (intervalo 0 a 359 grados) y en los botones ‘Left’ y ‘Right’ para decrementar o incrementar el empuje respectivamente (intervalo -100 a 100). La Figura 9.6 muestra un ejemplo de interactuación con estos botones en el que se ha hecho clic 13 veces en los botones ‘CW’ y ‘Right’.

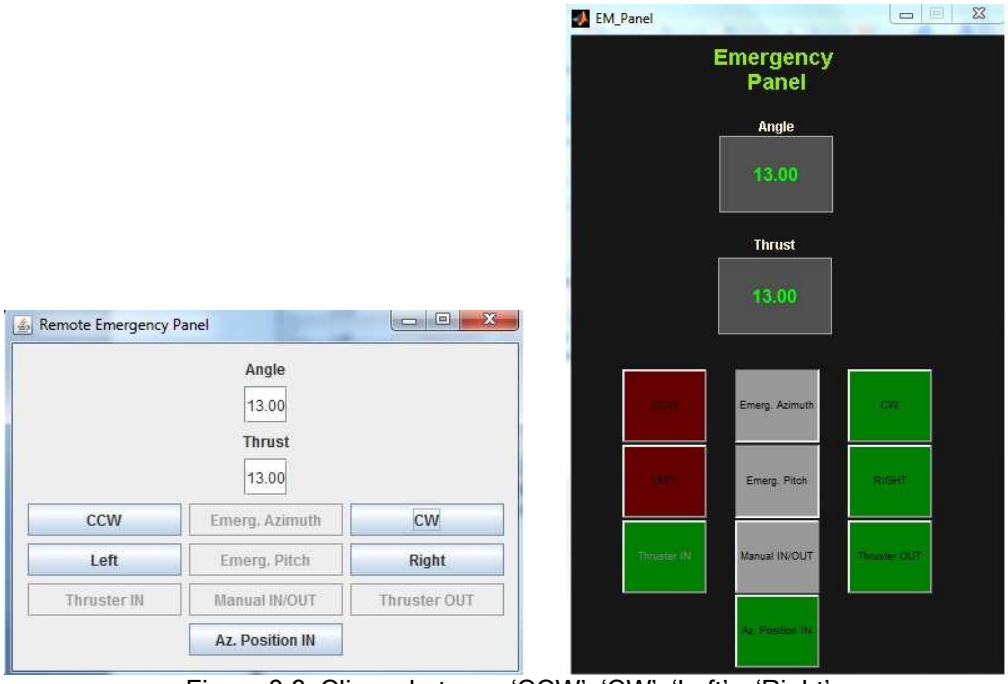


Figura 9.6. Clic en botones ‘CCW’, ‘CW’, ‘Left’ y ‘Right’.

7. Para que al pulsar en el botón ‘Az. Position IN’ se habilite el botón ‘Thruster IN’ es necesario que el ángulo y el empuje sean cero. Jugar con los botones ‘CCW’, ‘CW’, ‘Left’ y ‘Right’ hasta conseguirlo. Posteriormente hacer clic en el botón ‘Az. Position IN’, con lo que se llega a la situación de la Figura 9.7. Al hacer clic en el botón ‘Thruster IN’ automáticamente se cierra la GUI de Matlab y, con ello, se para la ejecución del programa del simulador del Panel de Emergencia del *Bow Thruster.* Finalmente cerrar la GUI para finalizar la simulación.

Figura 9.7. Clic en botón ‘Az. Position IN’.

Capítulo 10. Conclusiones y Trabajos Futuros

10.1. Resumen

Para llevar a cabo este PFM (Proyecto Fin de Máster) denominado “Control de vehículos con CompactRIO. Integración de aplicaciones Matlab y Java” se han seguido las siguientes fases de diseño:

1. En la primera fase de diseño se han investigado toolboxes de comunicaciones TCP/IP así como de generación XML en Matlab. Además se ha investigado en librerías de generación XML en Java. En los capítulos 4 y 7 se describe el resultado de sendas investigaciones.

2. En la segunda fase de diseño se ha desarrollado el *software* de ambas aplicaciones. Para diseñar e implementar una API (*framework*) que cumpla con las características de fácil ampliabilidad, extensibilidad, distribuibilidad... se ha empleado el patrón de diseño Objeto Activo descrito en el apartado 2.4 del contexto de desarrollo y así como en el Capítulo 6 de este documento. Además se ha organizado el código de control del Panel de Emergencia del *Bow Thruster* realizando un diseño en niveles de la API tal y como se describe en el Capítulo 7. Para que cada una de las partes del código desarrollado sean perfectamente legibles y fácilmente modificables se ha comentado el código utilizando la herramienta *Javadoc* que proporciona el entorno de desarrollo Eclipse tal y como se indica en el Capítulo 8 donde se ha hecho especial hincapié en la forma empleada para implementar las comunicaciones y generar/interpretar los comandos XML en Java y Matlab

3. En la tercera fase de diseño se ha depurado el *software* desde una estación de trabajo donde se han ejecutado ambas aplicaciones. Tras interactuar con las GUIs desarrolladas en ambos lenguajes se ha descrito un manual de usuario que se puede consultar en el Capítulo 9.

10.2. Conclusiones

En este apartado se van a analizar los resultados obtenidos a lo largo del proyecto de acuerdo con los objetivos planteados al inicio del mismo. En primer lugar se han investigado toolboxes de comunicaciones TCP/IP así como de generación XML en Matlab. Además se ha realizado la búsqueda de librerías de generación XML en Java. Hasta este punto se han cumplido con los requisitos investigadores asociados al PFM.

Posteriormente se han realizado pruebas de comunicaciones entre un proceso Matlab y una aplicación Java cumpliendo con otro de los hitos del proyecto. A continuación se ha definido una aplicación de prueba consistente entre un simulador Matlab y una aplicación Java. Para probar este escenario se ha definido un conjunto de instrucciones (comandos) que permiten controlar el simulador implementado.

Finalmente se ha implementado una aplicación de teleoperación en Java y se ha elaborado una memoria descriptiva del proyecto.

Para concluir este apartado decir que se han alcanzado cada uno de los objetivos fijados al principio del Proyecto Fin de Máster.

10.3. Trabajos Futuros

Para finalizar vamos a proponer a groso modo algunas líneas de investigación interesantes relacionadas con el Proyecto Fin de Máster que se ha llevado a cabo. Este proyecto puede servir de base para establecer una plataforma de pruebas del prototipo de vehículo con unidad de control CompactRIO que tiene el grupo División de Sistemas e Ingeniería Electrónica (DSIE) de la Universidad Politécnica de Cartagena (UPCT). El objetivo a alcanzar es un SW HMI (*Human Machine Interface*) que permita controlar y monitorizar dicho prototipo. Para ello la interfaz hombre-máquina implementará funcionalidades para arrancar, parar, acelerar, frenar, establecer la velocidad, girar a la izquierda, girar a la derecha, etc., además mostrará indicaciones de arrancado, parado, velocidad actual, posición etc. Este Sistema de Gestión Vehicular permitiría el monitoreo y rastreo del prototipo de vehículo de una manera eficiente y proveerá de herramientas efectivas para la operación, administración y control del prototipo. Por esta razón permitirá tomar decisiones óptimas en cuestión de fracciones de segundo. La Unidad de Control CompactRIO ubicada en el interior del prototipo reportará información de control que será actualizada constantemente, siendo posible monitorizar su posición, velocidad y estado en tiempo real. Estos datos serán procesados, almacenados y visualizados en un mapa digital para una rápida y fácil interpretación.

Capítulo 11. Referencias

[1] http://www.ni.com/compactrio/esa/

[2] Douglas Schmidt et al. "*Pattern Oriented Software Architecture*", Vol II, John Wiley and sons Ltd 2000.

[3] E. Mandado, J. Marcos, C. Fernández, J. I. Armesto, S. Pérez. "Autómatas Programables. Entorno y Aplicaciones". Editorial Thomson, 1ª edición, 2006.

[4] P.A. Lee y T. Anderson. "Fault tolerance: principles and practice". Editorial Springer-Verlag, 2ª edición, 1990.

[5] B. W. Ball y S.R. Cole. "*Design principles for safety systems*". Instruments Society of America (ISA). Transactions, vol. 30, nº4, pp. 9-18, 1991.

[6] B. W. Johnson. "*Design and analysis of fault tolerant digital systems*". Editorial Addison -Wesley, 1989.

[7] J. J. Rodríguez-Andina. "Métodos de síntesis de sistemas electrónicos seguros ante averías realizados mediante circuitos digitales configurables". Tesis doctoral Departamento de Tecnología Electrónica. Universidad de Vigo, 1996.

[8] J. M. Angulo, I. Angulo y J. García. "Fundamentos y estructura de computadores". Editorial Thomson Learning Paraninfo, 2003.

[9] P. de Miguel. "Fundamentos de los Computadores". Editorial Thomson Learning Paraninfo, 2004.

[10] E. Mandado. "Sistemas Electrónicos Digitales". 8ª edición. Editorial Marcombo, 1998.

[11] http://www.mathworks.es/products/instrument/

[12] http://www.geodise.org/toolboxes/generic/xml_toolbox.htm

[13] http://jdom.org/