UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

CONTROL Y AUTOMATIZACION DE UN SIMULADOR DE CIRCULACION CEREBRAL

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERO MECANICO ELECTRICISTA

P R E S E N T A N :

DAVID GUTIERREZ RUIZ
DANIEL KORNHAUSER EISENBERG

DIRECTOR DE TESIS: ING. FRANCISCO JAVIER CARDENAS FLORES
CO-DIRECTOR: DR. FABIAN GARCIA NOCETTI

MEXICO, D. F. MARZO 1997

Agradecemos

A la Universidad Nacional Autónoma de México.

A la Facultad de Ingeniería.

Al Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas.

Al Dr. Fabián García Nocetti, Jefe del Departamento de Electrónica y Automatización del IIMAS, por el apoyo brindado para la realización de esta tesis.

Al Ing. Martín Fuentes Cruz, por su ayuda técnica en la elaboración de la parte de electrónica.

Al Dr. Juan Nader Kawashi, autor de la idea de esta tesis.

Y un agradecimiento muy especial a nuestro director de tesis y además amigo por siempre

Ing. Francisco Javier Cárdenas Flores

Dedico esta tesis :

A mi padre,
por enseñarme el valor del trabajo y el esfuerzo.

A mi madre,
por su gran ejemplo de superación y entrega.

A mis hermanas,
con el deseo de que pronto experimenten la misma alegría que esta tesis me causa.

A mis amigos de la escuela, del IIMAS y los toros,
por compartir los buenos y malos ratos.

A Daniel,
por compartir parte de sus conocimientos conmigo.

David Gutiérrez Ruiz

Dedico esta tesis a :

Mi madre Rose,

que me ha enseñado cuales valores conforman un espíritu académico.

Mi padre Luis,

que me enseño mi primera computadora, y me explicó porque eran "tontas".

Mi hermano Adrián,

por compartir mis frustraciones en nuestro recorrido diario por la ciudad de México.

David,

por haberme enseñado a trabajar en grupo.

Agradezco a

La Mancha (alias Fernando Magariños Lamas)
La Maestra Lupita Ibargüengoitia.
La Maestra Susana García Salord.
La bola de personajes extraños que se reúnen a diario en el laboratorio y en todo el IIMAS.

Daniel Kornhauser Eisenberg

Indice General

Capítulo 1 : Introducción.

1.1 Definición del problema

Durante la década pasada, la bioingeniería logró notables avances. Esta ciencia emplea conceptos y métodos de ingeniería y computación para el estudio de los seres vivos. En sus aplicaciones prácticas, está relacionada con el diseño y construcción de instrumentos para la investigación biológica y la práctica en medicina. Etienne Jules Marey, pionero de la fisiología [1] , puso especial atención en actividades que muchos veían como modernas innovaciones, tales como el uso de las últimas herramientas de la física y la ingeniería. La siguiente declaración (hecha en 1878) fue anuncio de las aplicaciones que hoy en día tienen estas técnicas en las ciencias biológicas:

"Por la causa de la rigurosa experimentación, todas las ciencias dan una mano. Cualquiera que sea el objeto de estos estudios, tal como medir una fuerza o un movimiento, o un estado eléctrico o una temperatura, ya sea él - el investigador- un físico, un químico o un fisiólogo, él se vale de los mismos métodos y emplea los mismos instrumentos".

En el pasado, el ingeniero fue muchas veces llamado para convertir una idea en un instrumento clínico útil. El microscopio, el estetoscopio, el oftalmoscopio, y otros numerosos instrumentos son ejemplo de inventos que han abierto nuevos fronteras al conocimiento. Durante los últimos años, un gran número de nuevos instrumentos han sido desarrollados y ahora son usados en todas las especialidades médicas. Parece que, con el rápido incremento de las facilidades técnicas, los ingenieros se ven cada día más involucrados en todas las áreas de la medicina.

En ésta, gran parte de los conocimientos se obtienen en forma práctica: materias como anatomía, fisiología, y otras, requieren que el alumno se enfrente a problemas reales. La experimentación con cadáveres y animales intenta ser una solución al deseo de estudiar dichas situaciones. Sin embargo, muchas de las condiciones quedan lejos de acercarse a las reales.

Una zona que científicos del área médica estudian y de la que quisieran conocer más, es el cerebro. Su complejidad y fragilidad hacen de él un órgano de difícil acceso. Es imposible experimentar con él ya que su función es vital.

Ante esta situación, han habido infinidad de esfuerzos por desarrollar herramientas que permitan simular las situaciones reales a las que se encuentra expuesto el cerebro sin poner en riesgo la vida de ningún ser. Este es el caso de el trabajo expuesto en esta tesis en

donde, con el apoyo de investigadores del Instituto Nacional de Neurología y Neurocirugía, se diseñó lo que es la tercera generación del simulador de circulación cerebral. Este sistema permitirá a alumnos del área poder estudiar la forma en que la circulación en el cerebro se comporta ante distintas fallas, como las que pueden producir *trombos* al ocluir una arteria.

Así pues, en este trabajo se conjugan elementos de diseño de hardware y software que permiten crear un sistema de adquisición de datos y de control de actuadores que, controlados mediante una interface gráfica de usuario (GUI) en una computadora, da la oportunidad de simular distintas situaciones a las que se ve sujeta la circulación en el cerebro.

1.2 Objetivo

Diseñar un modelo físico para simular la circulación cerebral con fines de docencia. Este modelo deberá reproducir defectos a los que la circulación cerebral se ve sujeta, cumpliendo con los siguientes requerimientos:

- Dado un flujo líquido laminar y dicrótico que simula parte de las condiciones físicas que se encuentran en las arterias cerebrales, controlar el ritmo al que ésta fluye.
- Generar fallas (oclusiones) en las arterias de este sistema en diversos grados (0, 25, 50, 75 y 100%).
- Controlar el sistema mediante una computadora con una interface gráfica.
- Sensar la presión en puntos estratégicos del sistema con el fin de observar el impacto de las fallas en el comportamiento del sistema.
- Visualizar en la computadora los controles del sistema, así como el oscilograma de la presión sensada.

1.3 Metodología

Debido a que ya existía una versión previa de este sistema, el primer trabajo desarrollado fue el de estudiar sus elementos con el fin de determinar cuales de ellos serían reutilizados en la nueva versión. De igual forma se realizó una investigación que determinó

los avances a nivel mundial en sistemas relacionados con el que se pretendía desarrollar. Esto incluyó una búsqueda hemerográfica entre artículos médicos, así como búsquedas en Internet. Entre estos trabajos se encontraron investigaciones en las que, al igual que el desarrollado en esta tesis, se utilizaron elementos como: modelos de plástico para simular arterias, bombas que generaban flujos muy semejantes al sanguíneo, y además se realizaron mediciones de presión. Sin embargo, el objetivo primordial de éstas oscilaba entre investigaciones específicas del comportamiento del flujo en las arterias, hasta el monitoreo de la concentración de hemoglobina en el cerebro. Una descripción más detallada de estas investigaciones, así como de los elementos reutilizados de anteriores sistemas, se muestran en el siguiente capítulo.

De acuerdo con los objetivos y después del análisis del problema, se determinó desarrollar esta tesis en términos de la instrumentación virtual* , teniendo como objetivo final hacer un sistema amigable al usuario y que además fuera fácil de mantener y mejorar.

Las primeras pruebas realizadas estuvieron relacionadas con la comunicación entre el microcontrolador y la computadora. Para esto se utilizaron diversas interfaces seriales, tales como las herramientas de comunicación *PCBUG11* (MSDOS), *Terminal* (Windows) y *MINICOM* (Linux). Estas herramientas más tarde fueron sustituidas por una interface gráfica de usuario (GUI) que se programó específicamente para el sistema. Las pruebas de comunicación permitieron configurar el puerto serie para lectura y escritura manejándolo como un dispositivo de *UNIX*.

Una vez establecida la comunicación entre la computadora y el microcontrolador, se implementó un prototipo de hardware. Éste se diseño bajo una filosofía modular que permitiera intercambiar o agregar módulos dependiendo de las necesidades del usuario. Con el prototipo se realizaron las pruebas correspondientes al movimiento de motores, programación del ritmo de la bomba y visualización de los datos adquiridos por el convertidor A/D del microprocesador. Una vez comprobado el funcionamiento óptimo del hardware se diseñaron los circuitos impresos. Este trabajo se desarrolló con PROTEL para Windows, que es un paquete CAD específico para estos fines.

Para el diseño de software, se eligió el lenguaje de programación *Python*. Este lenguaje permite una programación orientada a objetos y da la facilidad de trabajar con módulos independientes que pueden ser editados fácilmente sin representar esto un cambio

* Este concepto se aborda posteriormente en el capítulo 3.

total. Python fue complementado con los módulos *Numeric*, *Tkinter (TCL/TK)* y *GIST* para la generación de la interface gráfica de usuario y la graficación de la señal digitalizada.

Para una buena cohesión y acoplamiento de problema el diseño se dividió en dos partes: La del componente del dominio de problema y la del componente de interacción con el humano, siguiendo la metodología de programación orientada a objetos de Coad.

De esta forma se crearon los objetos *Bomba*, *Monitor*, *Motores*, cada uno con su objeto puerto *Serial* independiente y por otro lado se creó una interfaz amigable para el manejo de estos objetos por medio de los widgets en los API de GUI TCL/TK*.

* Este concepto se aborda posteriormente en el capítulo 4.

Capítulo 2 : Antecedentes. Descripción de sistemas anteriores.

2.1 Desarrollos a nivel mundial

La tarea de simular un fenómeno físico para estudiar su comportamiento es la razón misma de la experimentación. Sin embargo, con el avance de la ciencia, los fenómenos a estudiar son cada día más complejos y difíciles de duplicar experimentalmente. Es por ello que investigadores de todo el mundo se han dado a la tarea de crear (con el apoyo de herramientas de hardware y software) instrumentos que, si no igualan al fenómeno físico que desean estudiar, si se aproximan en su comportamiento para las condiciones deseadas. Esto último es a lo que se llama *simulador*.

En el caso de esta tesis, la tarea de simular los efectos que oclusiones en las arterias del cerebro pueden causar a la presión sanguínea es un trabajo que se ha estudiado desde muchos puntos de vista. Como parte de la investigación previa realizada, se encontraron algunos proyectos relacionados con el desarrollado en esta tesis. Se busco entre artículos médicos con temas como la simulación de flujos sanguíneos en el cerebro , cálculo y medición de flujo y presión sanguínea en el cerebro con métodos no invasivos, elaboración de modelos físicos de arterias, etc.

Uno de estos trabajos fue el reportado por Brian W. Chong [1] en 1994. El propósito de este trabajo fue el de describir los patrones de flujo en un modelo de la arteria vertebrobasilar y usar estas observaciones para explicar la apariencia del flujo en imágenes obtenidas por resonancia magnética.

Este es el desarrollo que más se asemeja a esta tesis, ya que en él se reporta la preparación de arterias artificiales hechas de cera y elaboradas usando como molde arterias de cadáveres. El error en la dimensión con respecto a las arterias reales que reportan los investigadores es de aproximadamente 0.1%. Estos investigadores también desarrollaron un sistema con una bomba consiguiendo un flujo dicrótico y laminar, además que hacen circular una solución con características muy cercanas a la del fluido sanguíneo.

Sin embargo, su investigación se abocó a la observación y medición del flujo, utilizando como método la *angiografía* con resonancia magnética y grabaciones del flujo con video superVHS de 35 mm. Principalmente realizaron mediciones de flujo en diversas condiciones geométricas y ante la presencia de *aneurismas*.

Otro trabajo interesante fue el realizado por Dieter Liepsch [2] en el Instituto de Biotecnología en Alemania. Basado en el problema de que la *arteriosclerosis* es la principal

causa de infartos cerebrales y al *miocardio*, utilizó la anemometría Doppler para analizar el régimen de flujo que sigue la sangre en el ser humano. Para ello utilizó fluidos *pseudoplásticos* y *viscoelásticos* para simular la sangre, y modelos de vidrio y silicón para simular las arterias. En este desarrollo también fue utilizado un modelo hidrodinámico de circulación, aunque no se menciona nada acerca de sus características y desarrollo.

La tarea principal fue la de medir los parámetros del flujo tanto en las paredes de la arteria, como en el centro, además del cambio del número de Reynolds en ambos casos. Como resultado, observaron que en los modelos de vidrio los flujos en las paredes y el centro se defasaban en 45°, mientras que en los modelos elásticos no había desviación.

En el caso de Sandro Rossitti [3] , del Departamento de Neurocirugía de la Universidad de Gotemburgo, Suecia, el trabajo consistió en aplicar el principio del mínimo trabajo como una optimización paramétrica en el estudio del crecimiento y adaptación de los árboles arteriales.

En esta investigación se llegó a un modelo matemático en el cual se muestra que el flujo en una arteria esta relacionada en forma cúbica con el diámetro de la arteria. Esto permite conocer el flujo de una las ramas sólo conociendo el flujo en la arteria principal y los diámetros y ángulos con respecto a ella del resto de sus ramas.

Todas las mediciones se hicieron *in vivo* utilizando expedientes de adultos saludables y realizando regresiones basadas en el principio del mínimo trabajo, el cual establece que para mantener un flujo volumétrico estable en una arteria, la velocidad se incrementa si el diámetro del vaso diminuye.

Finalmente, la investigación realizada por C.E. Elwell [4] , aunque no se relaciona directamente con el tema de esta tesis, si fue una de las que mayores aplicaciones prácticas encontró en el área de la medición de flujos y presiones en arterias cerebrales.

En este artículo se reportan mediciones de flujo y presión arterial cerebral usando un diodo láser de cuatro longitudes de onda y un tubo foto multiplicador como método no invasivo de detección de variación en los niveles de $PACO_2$ en la sangre. Los niveles de $PACO_2$ se controlaban mediante la manipulación de FIO_2 y $FICO_2$. Estos gases son inhalados por el sujeto de medición (que en este experimento fue un cerdo, aunque en el siguiente reporte ya se realiza en humanos) y el tubo foto multiplicador detecta la respuesta del flujo y presión en la sangre ante cambios en los niveles de los gases.

Además de las investigaciones descritas, existen infinidad de desarrollos por computadora (empleando sólo herramientas de software) que en general utilizan sistemas multimedia de despliegue y algoritmos matemáticos para la simulación de los efectos y condiciones deseadas.

Como resultado de esta investigación de desarrollos a nivel mundial se puede decir que:

- El estudio de las condiciones de presión y flujo a las que se ve sujeta la circulación sanguínea en el cerebro es de gran interés tanto en situaciones específicas como en generales. En el caso expuesto en esta tesis, la presión se analizará desde un aspecto general en el que la variación de los factores de falla y condiciones propias de las arterias serán parámetros que el usuario manejará.
- El manejo de elementos automatizados permite realizar la simulación del problema de forma más precisa. La aplicación de tecnologías como la Resonancia Magnética, la Espectroscopía y el empleo de bombas para producir flujos laminares, son ejemplos de la forma en que la ingeniería da soporte a investigaciones dentro del área médica.
- Como punto en común entre todos estos desarrollos y el propuesto en esta tesis se encuentra el deseo de estudiar *métodos no invasivos* de medición de parámetros cerebrales. Por ello, el desarrollo de modelos que duplican las condiciones *in vivo* de anatomía y fisiología humana son de gran ayuda en el área médica.

2.2 Sistemas desarrollados en el Laboratorio de Patología Vascular

El denominado "Simulador de Circulación Cerebral" es un proyecto que lleva ya varios años de desarrollo. En un principio, un grupo de médicos del Laboratorio de Patología Vascular (Instituto Nacional de Neurología y Neurocirugía - INNN -), se dieron a la tarea de desarrollar con sus propios medios los diversos elementos que dicho simulador requería. Fue así como surgieron la bomba de flujo dicrótico y las llamadas primera y segunda generación de este proyecto.

2.2.1 Bomba de flujo dicrótico

Éste es el sistema hidráulico encargado de proporcionar un flujo laminar y dicrótico, tal y como lo hace el corazón en el ser humano. Esta compuesto de una bomba, dos capacitores hidráulicos, una válvula solenoide y una resistencia hidráulica.

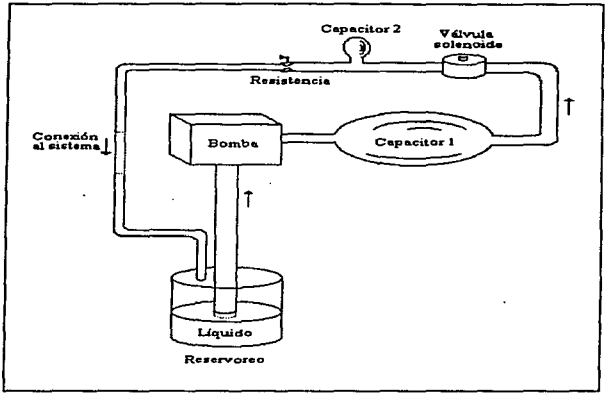


Figura 2.1 Bomba de flujo dicrótico.

Al comenzar el funcionamiento de la bomba (Figura 2.1), la válvula solenoide se encuentra cerrada. Bajo estas condiciones, el capacitor 1 se carga a su capacidad máxima, hasta el momento en que la válvula se abre, dejando escapar el flujo almacenado. Parte de este flujo pasa por la resistencia antes de ir al sistema; el resto del flujo llena al capacitor 2. Al llegar a su límite de capacidad, el capacitor 2 se descarga por la resistencia enviando su flujo al sistema. Este funcionamiento se repite al mismo ritmo al que la válvula abra y cierra, considerando que la bomba suministra un flujo constante. Una gráfica de este funcionamiento se muestra en la siguiente figura:

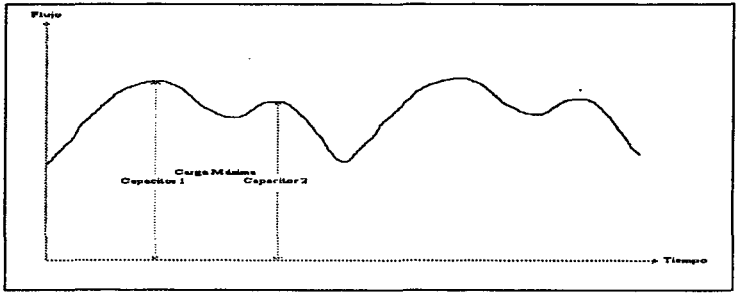


Figura 2.2 Flujo dicrótico.

2.2.2 Simulador de circulación cerebral (1ra generación)

Una vez implementado el sistema que proporcionaría el flujo requerido al sistema, se procedió al desarrollo de los modelos de las arterias. Estas fueron diseñadas en plástico delgado que proporciona características de forma y consistencia muy cercanas a las reales. La forma dada a estas arterias fue la del *Polígono de Willis* . Esto se logró cubriendo una película plástica sobre un modelo de cobre de dicha arteria.

Como sistema de oclusión se utilizaron electroimanes. Un diagrama del sistema se muestra a continuación:

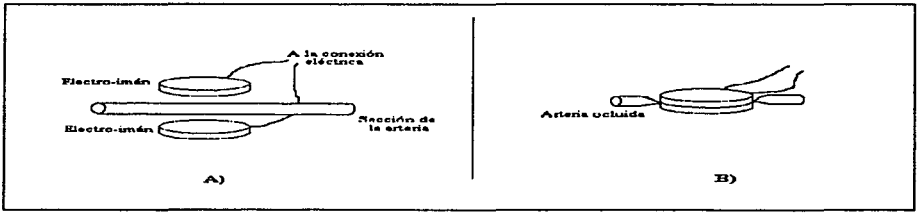


Figura 2.3 Sistema de oclusión controlado por electroimanes: A) Componentes; B) Al conectar a la corriente los electroimanes se produce la oclusión.

Se puede observar que la arteria es ocluida sólo cuando se le suminstra corriente a los electroimanes. En ese momento se atraen mutuamente presionando la arteria. Esto tiene la desventaja de que la oclusión sólo puede ser del 0 ó 100% (a diferencia del sistema implementado en posteriores generaciones). Otra desventaja notada en este sistema es el peligro de corto circuito y oxidación en caso de contacto entre los electroimanes y el líquido circulante.

Este sistema fue rápidamente desechado por las desventajas antes mencionadas, aunadas a su gran tamaño (que lo hacia difícil de manejar) y a su alto costo.

2.2.3 Simulador de circulación cerebral (2da generación)

Ya desarrollados los modelos plásticos de las arterias, se buscaron nuevas formas de lograr las oclusiones en estas. En la segunda generación se implementó un sistema hidráulico formado por dos jeringas conectadas por un tubo. Éste sistema se muestra en la siguiente figura :

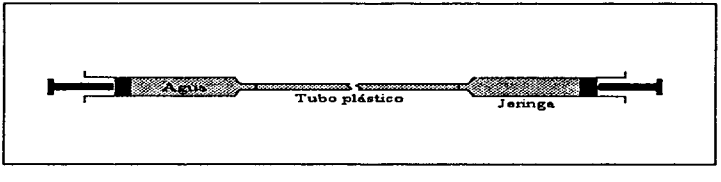


Figura 2.4 Sistema hidráulico de oclusión.

Al ejercer una presión sobre el líquido de una de las jeringas, se desplazará el liquido interno, produciendo un desplazamiento en el pistón de la otra jeringa.

Aprovechando este principio, se decidió colocar la arteria de plástico sobre una base de acrílico, dentro de la cual estaría una de las jeringas. La otra jeringa (fuera de la base) funcionaria como manipuladora. Al empujar o jalar el pistón de la jeringa manipuladora se produce un ascenso o descenso en el nivel del pistón de la otra jeringa, presionando o liberando a la arteria. Esto se ejemplifica en el siguiente diagrama:

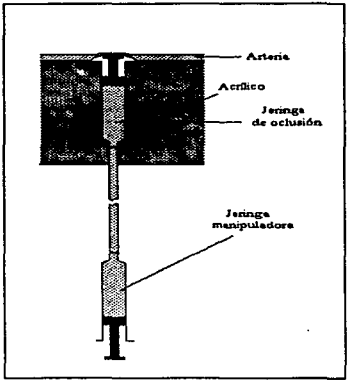


Figura 2.5 Sistema hidráulico de oclusión: base de acrílico.

Para lograr que la arteria se ocluya es necesario que se encuentre perfectamente adherida a la superficie de acrílico, así como que las jeringas estén perfectamente selladas y *purgadas* para que el desplazamiento hacia arriba y abajo sea lineal (así como la oclusión).

Este sistema fue de gran éxito debido a sus características:

- Debido a que el movimiento es trasmitido hidráulicamente, la oclusión es proporcional a la presión ejercida sobre la jeringa manipuladora.
- El sistema ocupa poco espacio, lo que permitió lograr diversas oclusiones en otras partes de la arteria.
- Bajo costo

Esto permitió diseñar una base de acrílico que contuviera ocho jeringas, con lo que se logran oclusiones en ocho lugares distintos de la arteria plástica, además de que ésta base acrílica se moldeó con la forma de un rostro humano, con lo que el simulador de circulación cerebral tomo su forma física definitiva.

Con este modelo fue con el que se trabajó en esta tesis para construir lo que sería la tercera generación del simulador de circulación cerebral.

2.2.4 Simulador de circulación cerebral (3da generación)

Conforme fueron avanzando los desarrollos en relación al Simulador de Circulación Cerebral, fueron aumentando los deseos de hacer de él un sistema automatizado. Los primeros esfuerzos fueron encaminados a sustituir el sistema manual de oclusión con uno que fuera controlado mediante un sistema mecánico manipulado desde una computadora.

Con la intervención de la computadora se abrieron las posibilidades de tener un sistema de adquisición de datos para muestrear señales de interés en el sistema. El trabajo con transductores de presión y un sistema de digitalización de señales llevó a los investigadores del INNN a respaldarse en el grupo que desarrolló esta tesis.

Bajo los objetivos planteados anteriormente, el simulador cobró el título de sistema de desarrollo en ingeniería. Así, se realizaron las modificaciones necesarias a la anterior generación de este sistema de modo que permitiera el muestreo de la presión en ocho puntos específicos del modelo plástico del Polígono de Willis. Se diseñaron los dispositivos de software y hardware necesarios, así como dispositivos mecánicos para la manipulación de los sistemas de oclusión. De igual forma, el sistema permite oclusiones en los ocho puntos distintos, concediendo al usuario simular distintas situaciones.

En los siguientes capítulos se explica la forma en que tanto hardware y software se desarrollaron para cumplir los objetivos planteados, así como las perspectivas para futuras generaciones del sistema.

Capítulo 3: Desarrollo del hardware.

3.1 Instrumentación Virtual

Como se ha podido observar hasta ahora, el problema planteado se puede solucionar mediante un sistema general de adquisición de datos y de control de actuadores. Este problema se puede clasificar dentro de la rama de la instrumentación llamada **Instrumentación Virtual** , debido al deseo de realizar un cierto control al sistema y mostrar los resultados de la adquisición realizada en una interface gráfica de usuario.

En el contexto de la electrónica, los instrumentos de medición tradicionales eran cajas negras a las cuales se conectaban señales de entrada y se recibía una salida que representaba un cierto análisis de la entrada.

Se pueden clasificar las funciones de un instrumento de medición tradicional en tres elementos - adquisición, análisis y presentación de datos -, tal como se muestra en la siguiente figura:

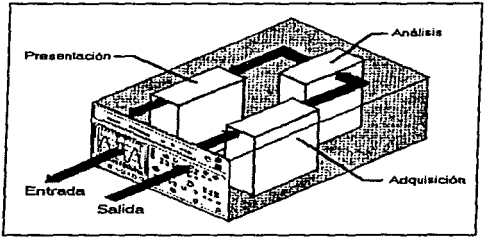


Figura 3.1 Instrumento de medición tradicional.

Un instrumento virtual también está compuesto de estos tres elementos, con la diferencia de que estos no siempre residen físicamente en la misma caja.

La evolución de las computadoras ha equipado a los usuarios con poderosas unidades de procesamiento y unidades de despliegue. Si aumentamos los beneficios de un instrumento de medición tradicional con la capacidad y flexibilidad de una computadora,

tenemos un instrumento virtual. Esto además, permite que el usuario defina el propósito de sus instrumentos a la medida exacta de sus necesidades.

Así pues, se puede definir un instrumento virtual como:

"el conjunto de software y/o hardware que junto con una computadora de propósito general permite a los usuarios interactuar con ella como si esta fuera su equipo de medición tradicional" [6].

Desde la estandarización del bus IEEE 488 (GPIB) y con la evolución de las computadoras, el uso de computadoras para el control de instrumentos se ha incrementado dramáticamente; día con día surgen nuevos elementos de hardware que conectados a una computadora (ya sea mediante un bus o un puerto) ofrecen conversiones analógicas-digitales, control de actuadores, etc. Así también, se ha ido incrementando el número de herramientas de software fáciles de usar. Esta paquetería incluye bibliotecas de elementos de instrumentación, bibliotecas para el análisis de datos y bibliotecas gráficas para una rápida construcción de interfaces gráficas de usuario. Se puede observar incluso el surgimiento de distintos módulos especializados para diversos lenguajes que generan estas interfaces en forma rápida y en pocas líneas de código.

Con todo esto, es posible construir diversas arquitecturas de instrumentos virtuales mediante la combinación de computadoras de propósito general con una variedad de hardware de instrumentación (Figura 3.2). Juntos, estos componentes interactúan entre sí para permitir la adquisición, análisis y presentación en la forma en que el usuario lo decida.

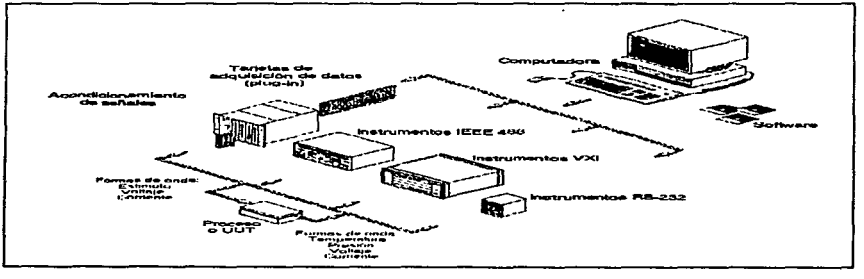


Figura 3.2 Elementos de instrumentación virtual.

3.2 Sistema de adquisición de datos y control de actuadores

Bajo la filosofía de un sistema de instrumentación virtual, se puede desarrollar un sistema general de adquisición de datos y control de actuadores basado en una computadora. La función principal de este sistema es la de medir y generar señales físicas que interactuen en un sistema real. Los componentes de un sistema de este tipo se observan en la figura 3.3 .

Los transductores sensan fenómenos físicos y los convierten en señales eléctricas tales como voltaje o corriente. Ejemplos de estos transductores son: termopares (temperatura a voltaje), RTD's (temperatura a resistencia eléctrica), galgas de esfuerzo (- strain gauges -, esfuerzo mecánico a voltaje), micrófonos (sonido a voltaje), etc.

Los dispositivos de acondicionamiento de señales pueden amplificar señales de bajo nivel, aislarlas y filtrarlas para una mayor precisión y seguridad de las mediciones. Estos dispositivos pueden también excitar y linealizar algún tipo de transductor.

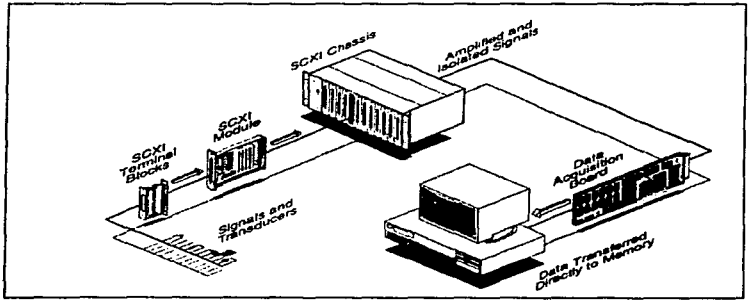


Figura 3.3 Componentes de un sistema de adquisición de datos y control de actuadores.

Una vez acondicionadas, las señales viajan a una interface de adquisición de datos para su digitalización. Por otro lado, esta interface genera las señales necesarias para el control de los actuadores. La programación del microcontrolador permite llamadas de funciones que se pueden usar con lenguajes convencionales de programación o paquetes comerciales.

La computadora determina la velocidad de procesamiento de la aplicación: aplicaciones que requieren procesamiento en tiempo real o de señales de alta frecuencia necesitan de un procesador de alta velocidad con su respectivo co-procesador o un procesador específico.

Bajo esta arquitectura, se diseñó el siguiente sistema que da solución al problema de simulación de la circulación cerebral bajo los requerimientos establecidos anteriormente:

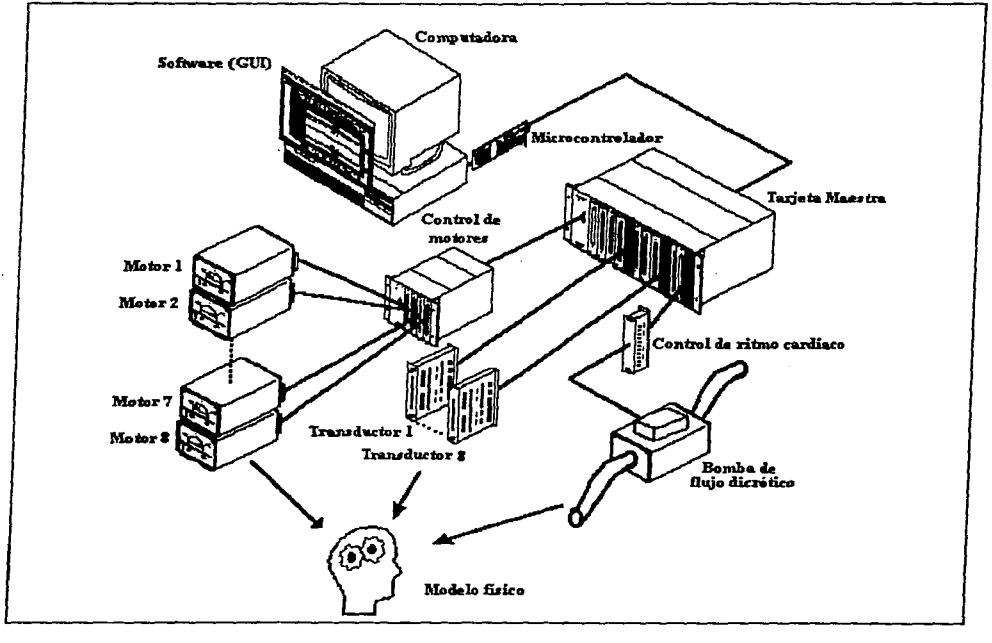


Figura 3.4 Sistema de Simulación de Circulación Cerebral (Diagrama de componentes).

A continuación, se explica el funcionamiento de cada uno de sus componentes relacionándolos con el modelo general de adquisición de datos y control de actuadores (explicado anteriormente). Una descripción más detallada de cada uno de estos elementos será expuesta posteriormente.

3.2.1 Transductores de presión

En el sistema desarrollado, los transductores de presión son de tipo *piezoresistor*, que generan un cambio en el voltaje de salida con variaciones en la presión aplicada. El elemento resistivo (una galga de esfuerzo) esta implantado iónicamente en un delgado diafragma de silicón. Aplicando presión al diafragma se produce un cambio en la resistencia de la galga, que igualmente produce un cambio en el voltaje de salida directamente proporcional a la presión aplicada (Figura 3.5).

El transductor utilizado en este sistema mide presión absoluta, tal como la presión barométrica, y la mide con respecto a un vacío absoluto ya integrado a ella. Este tipo de transductores son utilizados en hospitales para el monitoreo de presión sanguínea en modo invasivo. El transductor se encuentra integrado en un empaque plástico que permite su conexión con elementos médicos, tales como sueros, llaves de paso, conexión para monitoreo electrónico, etc.

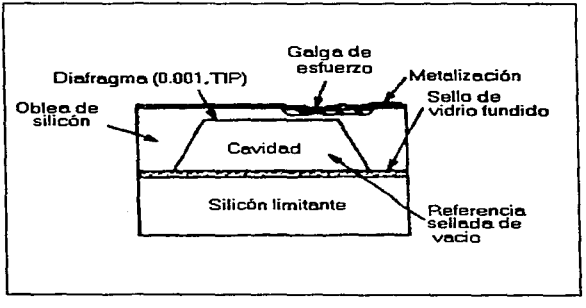


Figura 3.5 Diagrama de sección transversal del transductor.

Eléctricamente, el transductor utilizado proporciona una salida de 0.1mV por una presión aplicada de 1 mm de Hg* . Se emplea una polarización de ±12 V y mantiene una salida lineal hasta los 250 mm de Hg (valor máximo al que podría llegar la presión arterial humana).

Las características enunciadas anteriormente fueron resultado de una caracterización que se realizó para obtener las condiciones óptimas del funcionamiento del transductor. Se probaron tres polarizaciones : ±5, ±12 y ±15 volts; bajo cada una de ellas se aplicaron presiones entre 0 y 240 mm de Hg, a un intervalo de 20 mm de Hg. Los resultados se muestran en la siguiente gráfica:

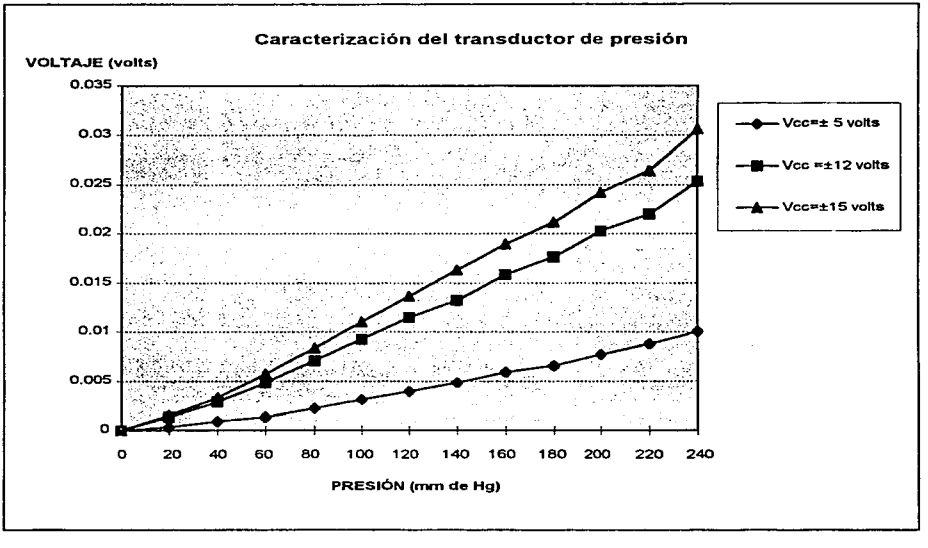


Figura 3.6 Transductor de presión: curvas de desempeño.

* Hg: Símbolo químico del Mercurio

Como se observa, cada una de las curvas mantiene distintos intervalos de no linealidad, lo que hace difícil la labor de determinar cual de ellas representa el funcionamiento óptimo. Debido a esto, con los datos obtenidos se realizó una regresión exponencial para cada una de las curvas.

Si llamamos **P** a la presión (en mm de Hg) ejercida sobre el transductor, y V_x al voltaje de salida para una determinada polarización, se tiene:

$$V_{\pm 5} = C_1 P^{1.2132}$$
$$V_{\pm 12} = C_2 P^{1.191}$$
$$V_{\pm 15} = C_3 P^{1.388}$$

donde C_1, C_2 y C_3 son constantes de relación con valores 3.9695×10^{-5}, 3.7×10^{-5} y 5×10^{-6}, respectivamente.

De estas ecuaciones se observa que, el exponente más cercano a 1 (condición de linealidad) es el que se logra con una polarización de ±12 volts. Fue por ello que se decidió utilizar esta condición.

3.2.2 Acondicionamiento de la señal adquirida

A cada uno de los transductores corresponde un circuito que nos permite acondicionar su señal de salida. éste consta de:

- Filtrado
- Amplificación : Permite llevar el voltaje de salida de los transductores a niveles entre 0 y 5 volts, que son los niveles estándar de entrada al convertidor A/D del microprocesador.
- Control de offset: Cada uno de los componentes de éste circuito tiene niveles de voltaje indeseables propios de su naturaleza. Es necesario eliminar estos voltajes para lograr una medición veraz.
- Protecciones: Dispositivos que protegen a los demás elementos del sistema en caso de fallas.

En términos generales, podemos ilustrar el funcionamiento de estos circuitos con el siguiente diagrama a bloques :

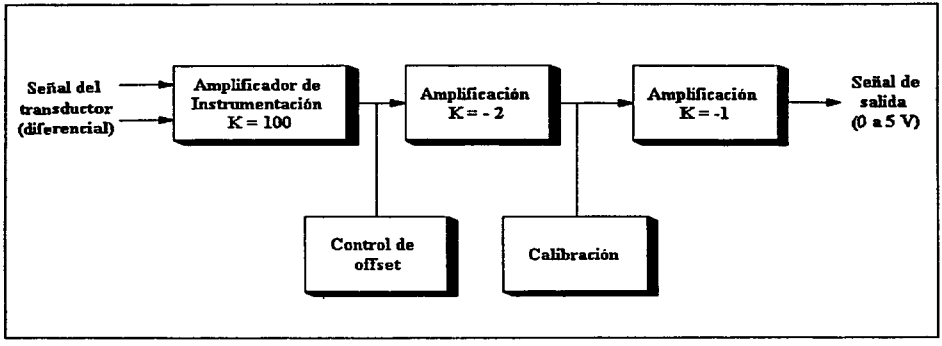


Figura 3.7 Diagrama a bloques del circuito de acondicionamiento de la señal.

3.2.3 Circuito de control de motores

Este circuito es el encargado de interpretar la lógica de la tarjeta maestra en acciones físicas que controlan el funcionamiento de un determinado motor a pasos. Es, en cierta forma, el enlace entre la electrónica digital y la lógica de potencia necesaria para el control de los motores. Detalles de su diseño se hablarán más adelante cuando se explique el control de la oclusión.

3.2.4 Tarjeta Maestra

La Tarjeta Maestra es la encargada de realizar la lógica necesaria de acuerdo a los datos que recibe del microcontrolador, para el control de los motores a pasos. Además, distribuye la potencia al resto del sistema en distintos niveles de voltaje, y contiene en forma física el bus donde se conectan cada uno de los circuitos de acondicionamiento para que estos entreguen sus salidas al microcontrolador. En la siguiente figura se muestran sus componentes :

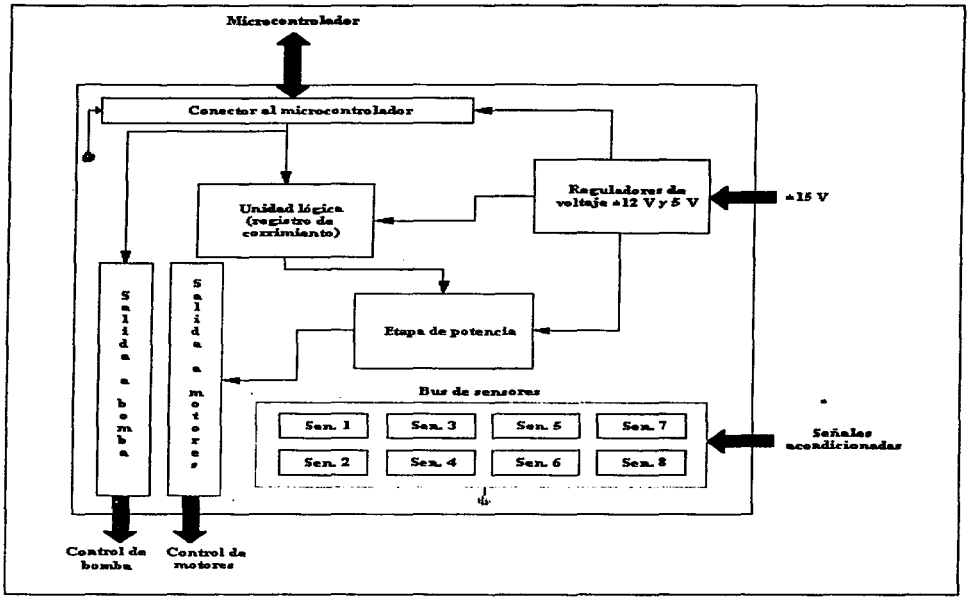


Figura 3.8 Diagrama a bloques de la tarjeta maestra.

3.2.5 Circuito de control de ritmo cardiaco

Tiene la función de controlar la apertura de la válvula solenoide (componente principal de la bomba de flujo dicrótico) permitiendo simular distintos ritmos a los que el flujo circulará en el sistema. Este es un sistema que maneja lógica TTL y potencia media, ya que maneja la línea de suministro eléctrico mediante el encendido digital de un triac. Dicho encendido a su vez es controlado por un temporizador programado por el microcontrolador. Más adelante se darán más detalles con respecto al diseño y funcionamiento de este circuito.

3.2.6 Microcontrolador

Un microcontrolador es un dispositivo de los llamados *computer on a chip*, ya que cuenta con una unidad aritmética-lógica, una unidad de control, unidades de entrada y salida, y memoria, todas ellas integradas en un mismo chip (figura 3.9). Para fines de este proyecto, se utilizó el microcontrolador MC68HC11E9 de Motorola, que cuenta con las siguientes especificaciones:

- 12 Kbytes de EPROM.
- 512 Bytes de EEPROM.
- 512 Bytes de RAM.
- Sistema de timer de 16 bit.
- Acumulador de 8 bit.
- Circuito de interrupción en tiempo real.
- Interface serial Síncrona.
- Convertidor analógico-digital de 8 canales a 8 bit.
- 36 pines de propósito general entrada o salida: 16 bidireccionales, 11 sólo entrada y 11 sólo salida.

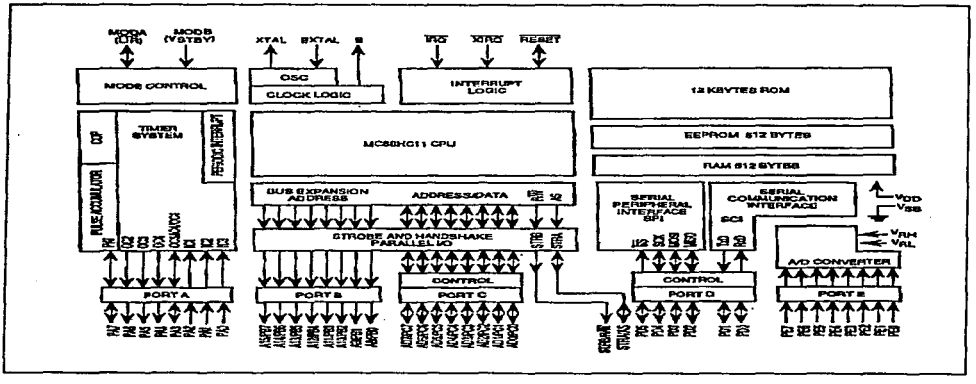


Figura 3.9 Diagrama de bloques del MC68HC11E9.

Para un mejor aprovechamiento y fácil manejo de estas características, Motorola creó la tarjeta de evaluación MC68H11EVBU (figura 3.10) que además de las anteriores características propias del microcontrolador, ofrece:

- Monitor BUFALLO , que contiene herramientas de comunicación, ensamblado (programación en código de máquina), manejo de sub-rutina propias, etc.
- Comunicación RS232C compatible con el puerto serial.
- Dispositivos de interface con otros elementos, como la incorporación de memoria externa, actuadores, transductores, etc.
- Requiere de una sola fuente de alimentación de 5 V.

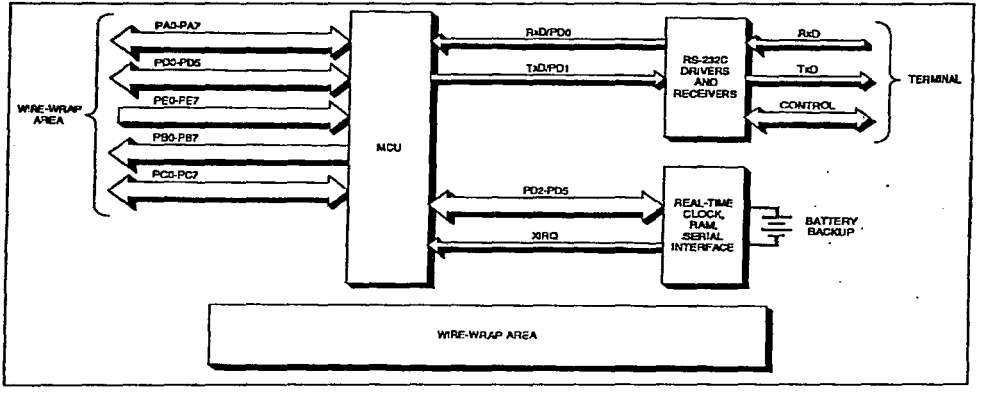


Figura 3.10 Diagrama a bloques del EVBU.

Dentro del sistema, el microcontrolador realiza la tarea de ejecutar programas (residentes en su memoria EEPROM) que producen salidas hacia la tarjeta maestra para que ésta realice la lógica correspondiente. La ejecución de dichos programas es controlada desde la computadora; es vía serial y mediante BUFALLO como se enlaza con el microcontrolador. También, el microcontrolador se encuentra encargado de la digitalización de las señales debidamente acondicionadas (proporcionadas por los transductores).

Así pues, se puede decir que el microcontrolador es el interprete entre las instrucciones de la computadora y la lógica que controla al sistema.

3.2.7 Computadora y software

La computadora es la vínculo entre el sistema y el usuario que, mediante una interface gráfica de usuario, manipulará el sistema. El software de dicho sistema esta diseñado para trabajarse sobre una plataforma LINUX. Esto nos ofrece múltiples ventajas, entre las cuales podemos encontrar el que es de distribución gratuita, permite el trabajo en red, es estable y es compatible con cualquier UNIX.

Para la programación fue utilizado Python, que es un lenguaje orientado a objetos que nos proporciona transportabilidad a otras plataformas UNIX y permite el rápido desarrollo de prototipos. Módulos anexos están disponibles para aumentar las posibilidades de este lenguaje.

Debido a que el software es el centro de comando y la imagen hacia el usuario del sistema, tiene una importancia vital. Características específicas de él se tratarán en el siguiente capítulo.

3.3 Sistema Integral

Una vez descritos cada uno de los componentes del sistema, ahora se describirá la forma en que interactuan para lograr el funcionamiento de los distintos bloques. El sistema, en función de las actividades que desempeña, puede dividirse en tres bloques principales:

- Control de la oclusión.
- Control del ritmo cardiaco.
- Adquisición de la señal de presión.

Todos estos bloques son controlados por el microcontrolador y es éste el único que se comunica directamente con el software en la computadora. Esto se puede apreciar más claramente en el siguiente diagrama a bloques:

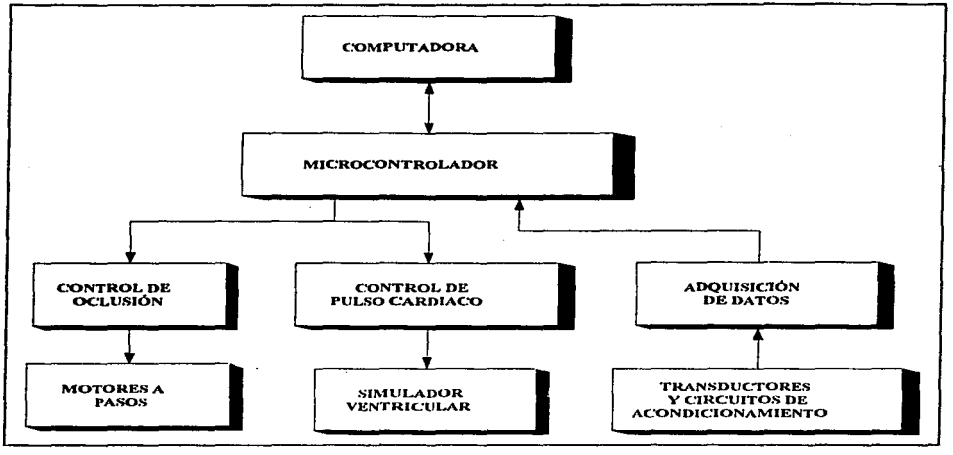


Figura 3.11 Sistema de Simulación de Circulación Cerebral (Diagrama a bloques).

A continuación, se describe el funcionamiento de cada uno de los bloques.

3.3.1 Control de oclusión

Uno de los requerimientos del simulador de circulación cerebral es el de producir fallas mediante oclusiones en puntos específicos de las arterias cerebrales (arterias de plástico que se encuentran dispuestas sobre el modelo físico). En la versión anterior del sistema este objetivo se lograba, en forma manual, mediante un sistema hidráulico de dos jeringas interconectadas entre sí por un tubo lleno de agua (tal como el usado en la generación anterior del simulador)* . Una de las jeringas funcionaba como manipulador, mientras que la otra se empleaba como pistón que, al subir o bajar, presionaba o liberaba la arteria.

* Ver el capítulo 2.2.2

En el sistema seguimos utilizando este método con algunas variantes, tales como:

- La jeringa manipuladora ya no es accionada manualmente, sino que es empujada o jalada por un *motor a pasos*.
- La jeringa actuadora ya no es un pistón, sino un gancho que al bajar presiona la arteria y al subir la libera.

Para lograr que el eje del motor a pasos empujara o jalara a la jeringa fue necesario introducir un sistema mecánico que transformara el giro del motor a pasos en un desplazamiento lineal. El sistema que resultó ideal para este efecto es el utilizado en los lectores de disco flexible de las computadoras para mover su cabeza magnética. Así, se sustituyó la cabeza magnética por el pistón de la jeringa y controlando el giro del motor mediante un circuito externo.

Existen dos tipos de motores a pasos, los de campo permanente y los de reluctancia variable. Los primeros normalmente tienen dos bobinas independientes, que en los llamados *motores unipolares*, tienen una derivación central (conocida como *tap central*). En el caso de los motores de reluctancia variable, normalmente tienen tres o hasta cuatro bobinas todas unidas en un punto común.

Los motores a pasos se pueden encontrar de distintas resoluciones angulares, que puede ir desde los 90° hasta los 0.72° por paso. Con un apropiado control, algunos motores de campo permanente pueden girar mitades de paso, y con mejores controles, se pueden obtener fracciones más pequeñas de paso, llamadas *micropasos*.

El motor utilizado en este sistema fue el de un *drive* de 5 ¼", unipolar, a 12 V, 0.16 A y 1.8 grados por paso. Este es un motor de campo permanente.

La configuración interna de sus bobinas es la siguiente:

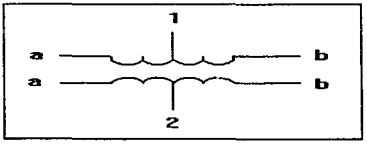


Figura 3.12 Motor a pasos (configuración de sus bobinas).

Para lograr el movimiento del motor se conectan sus tap centrales (1 y 2) al voltaje de alimentación (se recomienda utilizar un limitador de corriente) y las terminales 1a, 1b, 2a y 2b se conectan a la siguiente lógica positiva:

1a	1100110011001100
1b	0011001100110011
2a	0110011001100110
2b	1001100110011001
	tiempo ----->

Esta lógica se genera en la tarjeta maestra de la siguiente forma:

Un registro de corrimiento (tal como el 74194) es un circuito de cuatro bits que, mediante un reloj y una señal serial, proporciona la lógica deseada. Basta con definir, en sincronía con su reloj, cuales de las cuatro salidas serán unos lógicos. De acuerdo a la lógica necesaria para mover al motor, por cada cuatro ciclos de reloj se necesita que dos salidas consecutivas sean unos lógicos. Esto se ejemplifica en el siguiente diagrama:

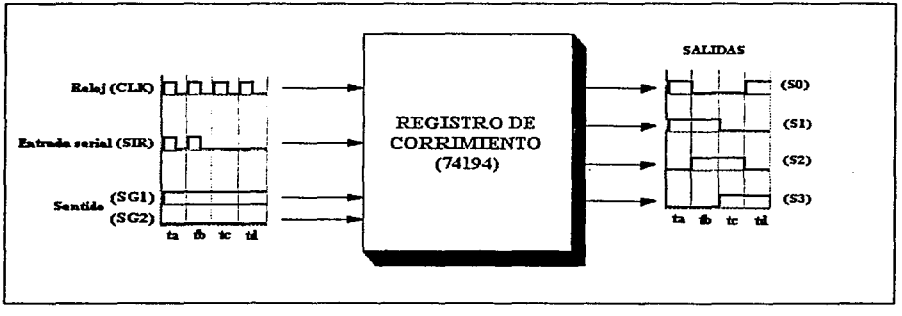


Figura 3.13 Funcionamiento del registro de corrimiento.

Los controles del sentido de giro (SG_1 y SG_2) son dos niveles lógicos que indican al registro si el sentido del corrimiento se iniciará en S_0 (bit menos significativo) o en S_3 (bit más significativo). Si $SG_1 = 1$ y $SG_2 = 0$ (como el caso mostrado en el diagrama), la secuencia irá de S_0 a S_3 mientras que, si $SG_1 = 0$ y $SG_2 = 1$, la secuencia irá de S_3 a S_0.

Los bits de control SIR, CLK, SG_1 y SG_2 son datos proporcionados por el microcontrolador (puertos A_7, A_6, A_5 y A_4 respectivamente). La salida de estos datos es controlada por un programa que reside en el microcontrolador el cual cíclicamente manda una secuencia completa de estos datos tantas veces como pasos del motor se desee. Este programa se realizó en lenguaje ensamblador y reside en la memoria EEPROM del microcontrolador.

Debido a las características del montaje del motor, el número máximo de pasos que puede dar antes de llegar al tope es de 20 . Debido a que el sistema debe realizar oclusiones del 0, 25, 50, 75 ó 100 %, el programa controla al motor para dar 0, 5, 10, 15 ó 20 pasos (en ambos sentidos). Así pues, se puede decir que el microcontrolador manda una secuencia completa de los bits SIR, CLK, SG_1 y SG_2 , 0, 5, 10,. 15 ó 20 veces, dependiendo de la oclusión que se le requiera al sistema .

Las salidas S_0 a S_3 pasan a una etapa de amplificación para convertir los niveles lógicos de 0 a 5 V a niveles 0 a 12 V. Esto se logra mediante el corte o saturación de 4 transistores TIP120 cuyas entradas son S_0 a S_3. La nueva lógica genera cuatro denominadas REL_1, REL_2, REL_3 y REL_4 (que alimentan las terminales 1a, 1b, 2a y 2b del motor a pasos)* .

Estas últimas señales son enviadas de la tarjeta maestra a la tarjeta de control de motores, en donde son dispuestas en un bus al que se conectan los ocho motores mediante ocho relevadores (de cuatro polos y dos tiros) que harán la función de interruptores para que sólo un motor trabaje a la vez.

De acuerdo a la configuración de las bobinas del motor a pasos (figura 3.12), las terminales 1 y 2 (tap centrales) se conectan directamente al voltaje de alimentación, mientras que las terminales 1a, 1b, 2a y 2b son conectadas a cada una a las entradas de los interruptores normalmente abiertos del relevador correspondiente. Las otras entradas de esos interruptores se conectan al bus. De esta forma se tienen 8 motores (con su correspondiente relevador) dispuestos en un bus . Mediante el accionar de los relevadores se elige cual de los motores se controlará (figura 3.14). La forma en que se accionan los relevadores se explica a continuación.

* Ver Anexo A. Figura A.2

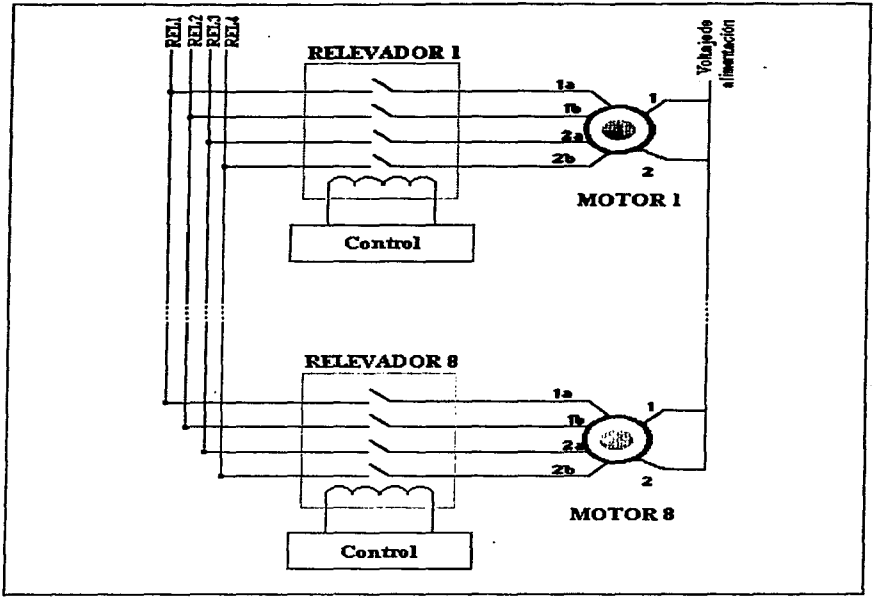


Figura 3.14 Arreglo de motores y conexión al bus.

3.3.1.1 Control de relevadores

El sistema de control asociado a cada uno de los relevadores no es más que un demultiplexor que, de acuerdo a la señal que el microcontrolador le envíe, elige entre los ocho motores el que estará activo durante la ejecución. Además cuenta con una opción de calibración, con la cual se lleva al motor activado hasta el tope de su carrera para después desactivar su funcionamiento, sin importar que el microcontrolador le ordene seguir moviéndose. A continuación se muestra el diagrama a bloques del funcionamiento de este sistema:

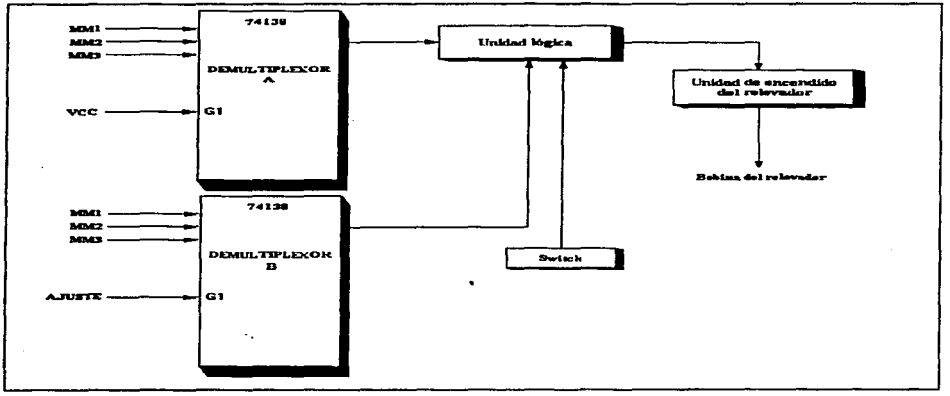


Figura 3.15 Control de encendido de relevadores.

El demultiplexor A es el encargado de accionar el sistema de encendido del motor que se desea mover. Esto es seleccionado mediante los bits MM_1, MM_2 y MM_3 que el microcontrolador (puertos D_2, D_3 y D_4 respectivamente) envía antes de iniciar el ciclo de comandos para el registro de corrimiento. El demultiplexor B indica si el motor activo se encuentra o no bajo la función de calibración. Esta calibración es física: se lleva al motor hasta el tope de su carrera; en ese momento acciona un switch que, mediante la lógica adecuada apaga al relevador. Dicha lógica (mostrada en el diagrama como una unidad lógica), es tal que, cuando el motor trabaja sin calibración, no toma en cuenta la acción del switch y cuando trabaja en calibración, la acción del switch es considerada. El demultiplexor B igualmente utiliza los bits MM_1, MM_2, y MM_3, además del bit AJUSTE (puerto B_4 del microcontrolador) que es el que acciona su funcionamiento. La llamada "unidad de encendido del relevador" no es más que la etapa de transistores que transforman la lógica 0 a 5 volts a una lógica de potencia 0 a 12 volts.

3.3.2 Control del ritmo cardiaco

Como mencionamos anteriormente, el flujo laminar y dicrótico a un determinado ritmo es proporcionado por una bomba. En el ser humano, el ritmo cardiaco puede variar desde 40 pulsaciones por minuto (*bradicardia*), pasando por niveles normales de 80 ó 90 , hasta valores de 200 (*fibrilación auricular*).

Con el fin de simular varias condiciones (tanto normales como extremas), se implementó un circuito electrónico que hiciera trabajar a la bomba a ritmos de 50, 60, 70, 90, 120 y 180 pulsaciones por minuto. Este circuito trabaja sobre la apertura de su válvula solenoide. El diagrama a bloques del sistema de control se muestra en la figura 3.16. Este es un sistema que trabaja en forma independiente del timer del microcontrolador. Esto es debido a que este último se encuentra realizando la digitalización de la señal al tiempo que la bomba opera.

El temporizador XR2240 es un componente que puede trabajar en forma monoastable o astable, con la particularidad que su período de oscilación puede ser programado mediante una palabra digital de hasta 8 bits. Esta palabra le indica al temporizador el número de veces que su base de oscilación ($T_b = RC$) será escalada. Esto es, si **n** es el valor decimal de la palabra binaria de programación y T_b es la base de período, entonces :

$T = (n + 1) T_b$, donde T es el período de la señal de salida.

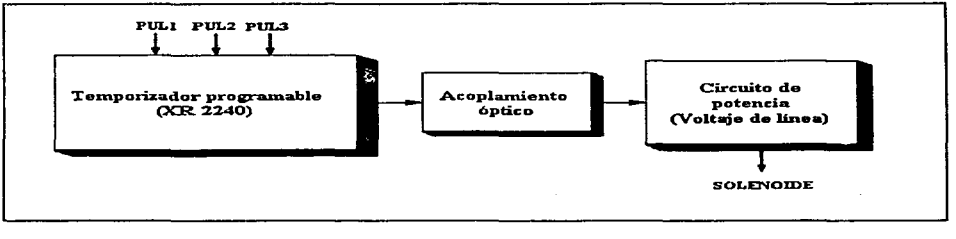


Figura 3.16 Control del simulador dicrótico.

Para el sistema, $T_b = 0.1667$ segundos y de acuerdo a los diferentes ritmos que se desean manejar, se obtuvo la siguiente tabla:

Palabra de programación $PULSO_3$	$PULSO_2$	$PULSO_1$	n + 1	T (segundos)	Pulsaciones por minuto
0	0	1	2	0.334	180
0	1	0	3	0.500	120
0	1	1	4	0.667	90
1	0	0	5	0.834	70
1	0	1	6	1	60
1	1	0	7	1.167	50

Los bits $PULSO_1$, $PULSO_2$ y $PULSO_3$ son enviados por el microcontrolador (puertos B_1, B_2 y B_3 respectivamente) para programar al temporizador. Cabe señalar que, aunque se pueden programar otros ritmos con palabras de hasta ocho bits, tres bits fueron suficientes para lograr los ritmos deseados.

La señal generada por el temporizador (señal cuadrada de 5 volts de amplitud) es llevada a un optoacoplador, con el fin de aislar la etapa digital de la de potencia (figura 3.17). Es necesaria una etapa de potencia debido a que el solenoide trabaja a voltajes de 110 V de alterna, consumiendo 9 W. La señal cuadrada, ya en la etapa de potencia, es la encargada de encender un triac (TIC206), el cual funcionará como switch del solenoide (encendiéndolo al mismo ritmo que el de la señal generada por el temporizador).

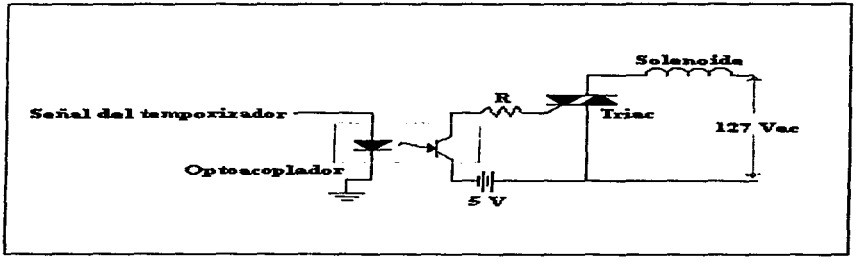


Figura 3.17 Acoplamiento óptico y circuito de potencia.

3.3.3 Adquisición de datos

Una vez dispuestas las condiciones bajo las que el sistema operará, es decir, después de haber elegido el tipo de oclusión y el lugar en donde se generarán, además del ritmo al que el flujo circulará en las arterias de nuestro modelo físico, la tarea del sistema es la de monitorear la presión en ocho puntos distintos de estas arterias.

La medición de la presión en las arterias se realiza mediante un método no invasivo: un pequeño anillo plástico lleno de agua sujeta a la arteria en el punto donde se desea hacer la medición. Este anillo hace contacto completo con la superficie de la arteria sin aprisionarla, de modo que la presión que el flujo circulante dentro de la arteria ejerce contra las paredes de ésta, se transmite al anillo plástico. Este último está directamente conectado al transductor de presión. Así pues, el flujo ejerce presión sobre la arteria, la arteria sobre el anillo y son las variaciones de presión en el líquido del anillo las que el transductor detecta (figura 3.18) .

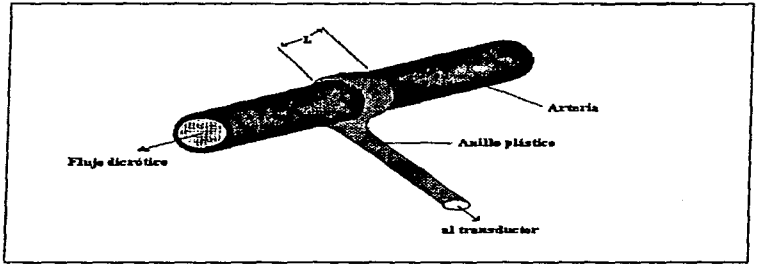


Figura 3.18 Sistema no invasivo de medición de la presión.

Este principio ya antes ha sido estudiado y, de acuerdo con L.A. Geddes [7] , debe mantener la siguiente relación dependiendo de la variación de volumen (ΔV) que el anillo sufrirá al variar la circunferencia (C_0) de la arteria ante la presión ejercida sobre sus paredes :

$$\Delta V = \frac{LC_0\Delta C}{2\pi}$$, donde ΔC es el incremento en la circunferencia de la arteria

Las variaciones de presión estarán en el rango de los 0 a 250 mm Hg, que son los valores que en el ser humano se pueden encontrar. De acuerdo a las especificaciones dadas en la sección 3.2.1, el transductor debe proporcionar una salida de voltaje en el rango de 0 a 25 mV. Sin embargo, el convertidor analógico-digital del microcontrolador trabaja con señales en el rango de 0 a 5 volts. Es por esto que después de adquirir el dato de presión y transformarlo a voltaje se necesita una etapa de acondicionamiento de la señal, la cual comprende filtrado y amplificado de la señal.

Este acondicionamiento inicia con un amplificador de instrumentación que proporciona una amplificación de 100 veces, además de quitar el ruido en modo común y de transformar la señal diferencial proporcionada por el transductor a una señal con referencia a tierra. Continúa con etapas de control de offset y otra etapa de amplificación, esta vez de 2 veces. Al final del acondicionamiento el sistema de adquisición de datos entrega al convertidor un voltaje de 0.02 volts por 1 mm Hg. Así, a los 250 mm Hg se tendrá un voltaje máximo de 5 volts.

Una vez acondicionada la señal, es entregada al convertidor del microcontrolador. Este es un convertidor de ocho canales (puertos E_0 a E_7) que, mediante una inicialización apropiada puede trabajar en distintos modos de conversión. En general, las operaciones de conversión A/D son realizadas en secuencias de cuatro conversiones cada vez. Una secuencia de conversión puede repetirse continuamente (modo SCAN activo) o detenerse después de una iteración (modo SCAN inactivo). También, se puede configurar al sistema para que éste realice cuatro conversiones consecutivas de un mismo canal (modo MULT inactivo) o que realice una conversión en cada uno de cuatro canales determinados - canales 0 al 3 ó 4 al 7 - (modo MULT activo).

El convertidor A/D realiza una conversión en 128 ciclos de reloj. Considerando el caso en el que, con un cristal de 2 MHz un ciclo es de 0.5 μs, una conversión completa toma 64 μs. Esto nos da una frecuencia de conversión de 15625 Hz.

Tomando en cuenta el teorema de Nyquist, el cual dice que la frecuencia de muestreo deberá ser por lo menos el doble de la frecuencia de la señal, obtenemos que la frecuencia máxima de la señal a muestrear puede ser hasta de 7812.5 Hz. Sin embargo, el microcontrolador siempre realiza cuatro conversiones antes de entregar los resultados (si MULT esta inactivo), con lo que esta frecuencia disminuye a 1953.125 Hz; o realizando 8 conversiones (MULT activo) la frecuencia disminuye a 976.5625 Hz.

Bajo todas estas condiciones, se trabajó bajo una configuración con SCAN y MULT inactivos, con lo que se obtiene el mejor desempeño en ancho de banda que el convertidor pueda proporcionar.

La forma en la que se trabajaron los datos obtenidos por la conversión es la siguiente: Mediante un programa residente en el microcontrolador se determina el canal que se desea muestrear, se guardan los resultados de estas conversiones en un buffer de 56 elementos y mediante la subrutina de comunicación serial de BUFALLO son enviados a la computadora. Esto se ejemplifica en la siguiente figura:

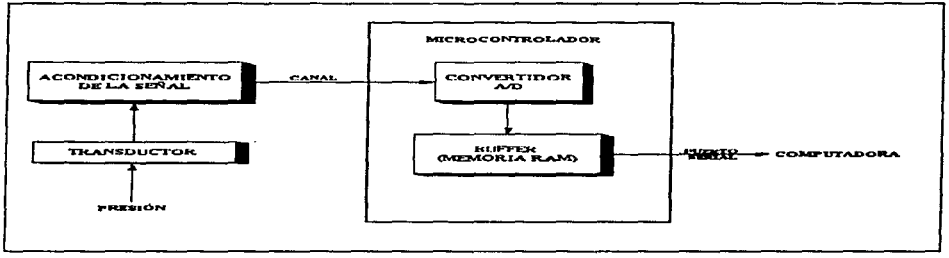


Figura 3.19 Adquisición de datos y digitalización.

La elección del canal a muestrear la realiza el usuario mediante el software. Esta acción genera un valor que es colocado en una localidad de memoria que el programa en el microcontrolador usará para programar al convertidor A/D. Los resultados de las conversiones son almacenados en memoria RAM para formar un buffer de datos que, al sumar 56, son enviados a la computadora vía puerto serial en forma de datos hexadecimales representados por dos caracteres ASCII.

Aunque el convertidor se encuentra trabajando en modo SCAN inactivo, éste realiza conversiones sucesivas del canal seleccionado debido a que el programa se mantiene en un ciclo continuo. De igual forma, el envío de los resultados a la computadora es continuo, y la operación de muestreo del canal sólo es detenida mediante un RESET.

Capítulo 4: Desarrollo del software.

4.1 Introducción

El software desarrollado tiene como propósito proveer una interface para configurar y observar el modelo físico del simulador de circulación cerebral. El sistema esta compuesto por dos bloques: El primer bloque controla al simulador ventricular y al sistema de oclusión arterial gracias a una interface gráfica. El segundo bloque despliega un monitor donde se observa el oscilograma de la presión en uno de los diferentes puntos de nuestro modelo físico.

4.2 Recursos necesarios para el diseño y uso del software

Para desarrollar el software se utilizó una computadora personal con un microprocesador 486DX a 33 Mhz y 12 M de memoria. El proyecto fue desarrollado en un sistema operativo con un kernel de Linux y basado en utilierías GNU . El lenguaje que se utilizó para desarrollar el sistema fue Python, con algunos módulos externos como Numeric, TCL/TK y Gist.

4.3 Descripción de las herramientas de software

A continuación se describirán las características del sistema operativo y del lenguaje utilizados para crear el sistema. Se mostrarán las ventajas que presentaban ante otras alternativas para desarrollar el sistema.

4.3.1 Sistema operativo Linux

Linux es un clon de Unix escrito por Linus Torvalds con la colaboración a través de la red de un grupo de hackers. Este sistema operativo tiene todas las características que uno puede esperar de un sistema Unix moderno: Es multitarea, tiene memoria virtual, usa bibliotecas compartidas, posee un manejo de memoria adecuado y provee la conectividad a red mediante el protocolo TCP/IP. Además el kernel Linux soporta todas las utilerías standard de Unix, y paquetería que se le ha portado como Emacs, X-Windows. La licencia

de Linux es de tipo libre (se distribuye el programa con sus fuentes), además de que existen distribuciones como el RedHat 2.02, que son muy flexibles y robustas.

Dado que la paquetería para el acceso a red es parte del sistema operativo, Linux posee todas las facilidades para conexión en red. Esto nos da la capacidad de controlar y observar procesos desde cualquier nodo de una red, utilizando utilerías incluidas en el sistema operativo.

Finalmente, como Linux es muy utilizado en ambientes científicos existe mucho software libre y gratis para analizar y procesar datos.

4.3.2.- Lenguaje de programación Python

Como lenguaje de programación se seleccionó el interprete Python. Es difícil clasificar este lenguaje con respecto a otros, dado que se utiliza en una gran variedad de aplicaciones. Por ejemplo, a veces se usa como un lenguaje de extensión empotrado, pero también se puede utilizar sólo, como una herramienta de programación independiente. Actualmente Python se emplea en los diferentes tipos de aplicaciones :

- Herramientas de shell
- Trabajo de Extensión de lenguaje
  - Front-ends a bibliotecas de C/C++
- Desarrollo rápido de prototipos
- Lenguajes basados en módulos
- Interfaces gráficas de usuario
  - Utilización de Simples y avanzados APIs de GUI
- Acceso a Bases de Datos
- Programación Distribuida
- Scripts de Internet

Las propiedades por las cuales se seleccionó fueron :

- Trabajo de Extensión de lenguaje
  - Front-ends a bibliotecas de C/C++
- Desarrollo rápido de prototipos
- Lenguajes basados en módulos
- Interfaces gráficas de usuario
  - Utilización de Simples y avanzados APIs de GUI

El lenguaje Python tiene un curva de aprendizaje muy pronunciada dado que es muy simple de utilizar. Por ejemplo, no se requiere declarar variables. Esta frágil tipificación permite al programador concentrarse más en un buen diseño del sistema que en su codificación.

Además, el manejo semántico de módulos permite una fácil extensión del lenguaje [8]. Por ejemplo, para resolver el problema de la interfaz gráfica, existen módulos como el Tkinter implementado en base al TCL/TK. Gracias a este módulo se puede crear rápida y fácilmente una interface gráfica de usuario (GUI) con botones, radiobotones y muchos otros objetos gráficos, que permiten al usuario un manejo sencillo e intuitivo de la aplicación. También Python funcionó perfectamente como front-end de bibliotecas de C como el Gist para implementar las aplicaciones que demandaban computo intensivo como la visualización de datos en tiempo real.

Finalmente, para el manejo eficiente de cálculos numéricos, Python tiene módulos escritos en C como el Numeric, para una rápida ejecución de los cálculos . Este módulo fue creado debido a que Python no es un vehículo adecuado para trabajos computacionalmente intensivos, por lo cual se diseñaron extensiones en C que permiten ejecutar operaciones mucho más rápidamente.

4.3.3.- Programación de la interface gráfica de usuario (TCL\TK)

a) Toolkit para programación de interfaces: TCL/TK

TCL/TK es una Interface para Programación de Aplicaciones (API) de Interfaces Gráficas de Usuario (GUI). Fue desarrollada por John Ousterhout en la Universidad de California, Berkeley. Este paquete se concibió para desarrollar interfaces gráficas fácil y rápidamente. TCL es el lenguaje de programación básico, mientras que TK es un toolkit de widgets, que son objetos gráficos similares a los de otros toolkits de GUI como Xlib y sdkwindows.

Se puede usar el TCL/TK en sistemas UNIX, DOS, Windows95 y Mac, aunque el uso extensivo de características del sistema operativo donde se desarrolla pueden volverlo menos portable. Por lo tanto es recomendable crear módulos fáciles de usar para después ligarlos con los comandos del sistema operativos que se este empleando.

El lenguaje TCL es normalmente interpretado, por lo tanto sus aplicaciones no se ejecutarán normalmente tan rápido como programas equivalentes en C. Pero para un amplio

espectro de aplicaciones , como el de esta tesis, no representa una desventaja, dado que la velocidad de procesamiento de las computadoras actuales es más que adecuada para un tiempo de despliegue razonable.

b) Módulo de acoplamiento de Python con TCL/TK: Tkinter

El módulo Tkinter fue desarrollado por Steen Lumholt y Guido Van Rossum. Este módulo acopla Python al TCL/TK. De esta forma se pueden desarrollar interfaces gráficas desde un programa de Python, con la misma rapidez y simpleza que en TCL/TK pero sin el rigor semántico de C.

4.3.4.- Biblioteca de graficación científica (Gist)

a) Biblioteca de graficación científica Gist

Gist es una biblioteca escrita por David H. Munro of Lawarence en el Laboratorio Nacional de Livermore. Maneja tres tipos comunes de despliegue de gráficas: X-Windows, Postscript (Color), y el ANSI/ISO standard de Computer Graphics Metafiles (CGM). La biblioteca es compacta (escrita directamente en Xlib), portable, eficiente y autónoma. Produce gráficas cartesianas, mesh quadrilaterales en dos dimensiones con contornos, campos de vectores y mapas de pseudocolor como meshes.

El modulo Gist de Python utiliza el modulo " Numeric " de J. Hugunin que esta implementado en C para manejar sus datos. Por lo tanto, se pueden manejar grandes cantidades de datos rápidamente. El módulo Gist incluye un despachador de eventos de X-windows que puede ser añadido dinámicamente al interpretador de Python.

b) Módulo de acoplamiento de Python con Gist : gist.py

El módulo de extensión gist.py fue escrito por Lee Busby de el Lawrence Livermore National Laboratory. Este módulo acopla a Python la biblioteca de graficación Gist escrita en C por medio de la cual uno puede desplegar gráficas de datos en un ambiente X-windows.

4.4 Especificación del software

La especificación es una parte critica dentro del diseño del software, dado que provee un punto de referencia a lo largo de su desarrollo. Por ello se formularon dos tipos de especificaciones : Primero se estableció la especificación los requerimientos del sistema para después formular la especificación del diseño del sistema. En la especificación de requerimientos, el usuario conviene con el diseñador las características de sistema. De este modo se plantea en una descripción informal que indica al diseñador como espera el usuario que el sistema reaccione cuando le dicte algún comando.

Una vez establecida la especificación de requerimientos, el arquitecto del sistema y el programador pueden trazar el diseño del software. En esta especificación se formula una descripción formal de proyecto, de tal manera que no exista ambigüedad en el comportamiento de cada uno de sus componentes.

4.4.1 Especificación de requerimientos

La especificación de requerimientos debe ser clara y comprensible para el usuario, por lo cual se utilizó el lenguaje natural para crear la siguiente especificación informal :

El sistema esta formado por dos bloques con los que el usuario puede interactuar de manera amigable mediante una interface gráfica de usuario :

- El primer bloque representa el control del modelo fisico que maneja la oclusión y el flujo en las arterias. Para ejercer la oclusión en la arteria deseada, el usuario puede seleccionar uno de los ocho motores y la posición que se desea proporcionarles . De esta forma se puede controlar la oclusión de la arteria en cinco grados : 0%, 25%, 50%, 75%, 100%, donde este último representa el número de pasos para alcanzar la oclusión total. El número de pasos que representa una oclusión del 100% puede ser determinada por el usuario mediante la opción de calibración. También se puede ajustar la oclusión de la arteria al 0% automáticamente. Para configurar el ritmo de la bomba de flujo diastólico el usuario sólo necesita selecionar uno de seis ritmos : 50, 60, 70, 90, 120, 180 pulsaciones por minuto.

- El segundo bloque esta compuesto de un monitor para observar la presión en uno de los diferentes sensores. Para activar el monitor el usuario selecciona que sensor de presión desea monitorear y por cuanto tiempo desea el monitoreo.

4.4.2 Especificación de diseño

La especificación de diseño debe de ser muy detallada para poder asistir a un buen diseño y una programación coherente de nuestro sistema. Por consiguiente se escribió una especificación formal, donde se usó una sintaxis preestablecida que eliminara toda posibilidad de confusión. Para este sistema se seleccionó escribir una especificación formal descriptiva lógica.

La especificación descriptiva lógica esta compuesta de fórmulas de teoría de primer orden (first-order theory FOT). Una formula de FOT es una expresión que implica variables, constantes numéricas, funciones, predicados y paréntesis, como en la aritmética tradicional. Utiliza los operadores lógicos tradicionales *and*, *or*, *not*, *implies* y = (que denota equivalencia lógica). El tipo de resultado de una formula de teoría de primer orden debe de ser booleano. Sin embargo a diferencia de muchos lenguajes de evaluación booleana, las fórmulas de FOT pueden también utilizar cuantificadores. Los símbolos *existe* y *para toda x* también pueden ser aplicados a las variables.

Para enunciar la especificación lógica se dividirá en el sistema de la siguiente forma :

- Sistema de control de la oclusión

Este sistema controla la posición de los émbolos hidráulicos. Dicho control se proporciona por medio del sistema de motores a pasos descrito en el capitulo anterior. Dado que este sistema se controla por motores de pasos no se verificará la posición del motor. El sistema contara sólo con tres rutinas de control :

- Rutina para ajustar en posición 0% la jeringa.
- Rutina para calibrar los pasos que tiene que dar el motor para ocluir 100%.
- Cálculo automático del número de pasos a dar para la siguiente posición según la posición actual.

- Sistema de activación de bomba

Este sistema envía una palabra al microcontrolador que activa al temporizador programable que controla la bomba diastólica.

- Sistema de monitoreo

Este sistema despliega en la pantalla una gráfica de las señales cuantizadas por el convertidor A/D del microcontrolador.

A) Especificación lógica del sistema de control de la oclusión

Identificadores :

M : Motor
S : Sentido (Podrá tomar los valores D : Derecha o I : Izquierda)
P : Pasos (Pmáx, número de pasos que conforma una carrera del émbolo)
BM : Botón Mover motor
BP : Botón de Posición
BC : Botón de Calibración
BA : Botón de Ajuste
BPA : Botón para mover un paso
SC : Sensor Calibración
t_o : Es el tiempo inicial
T :Tiempo cuando ocurre el evento

Eventos :

- Envío de comandos al microcontrolador para posicionar el motor M al presionar la tecla mover motor con la posición seleccionada :

mover motor(BM,M,BP,T)
BM en [true,false] , M en [1..8] , BP en [1..5] , $T \geq t_o$

- Envío de comandos al microcontrolador para mover el motor M un paso hacia adelante o atrás :

  mover un paso(BPA,M,P,S,T)
  BPA en [true,false] , M en [1..8] Pasos[1..20], S [D,I] , $T \geq t_0$

- Envío de comandos al microcontrolador para calibrar la carrera actual del motor M a una posición de 100 % al presionar la tecla "calibra motor" :

  calibrar motor(BC,M,T)
  BC en [true,false] , M en [1..8] , $T \geq t_0$

- Envío de comandos al microcontrolador para ajustar en la posición cero el motor M al presionar la tecla "ajustar a 0%" :

  ajustar motor(BA,M,T)
  BA en [true,false] , M en [1..8] , $T \geq t_0$

- Paro automático del motor cuando se presiona el sensor de posición :

  ajustado(SC,M,T)
  SC en [true,false] (abierto,cerrado), M en [1..8] , $T \geq t_0$

Estados :

- El motor M se esta moviendo en dirección D los pasos necesarios para llegar a la posición P :

  moviendo(M,S,BP, T_1, T_2)
  M en [1..8] , S en [derecha,izquierda] , BP en [1..5] , $t_0 \leq T_1 < T_2$

- El motor M se encuentra parado en la posición P :

  parado(M,BP, T_1, T_2)
  M en [1..8] , BP en [1..5] , hasta $t_0 \leq T_1 < T_2$

- Después de cada movimiento se guarda la posición P en la que el motor M se encuentra.

posición(M,BP, T_1, T_2)
M en [1..8] , BP en [1..5] , $t_0 \leq T_1 < T_2$

- El motor M se calibra con el número de pasos actual en la posición Pmáx (posición de 100%)

calibrado(M,P,BP, T_1, T_2)
M en [1..8] , P en [1..20] , BP en [5], $t_0 \leq T_1 < T_2$

Reglas de los motores que relacionan los eventos con los estados :

R_1. Cuando uno selecciona la posición P del motor M y presiona el botón de posición BP, éste se empieza mover.

mover motor(M, BP, T_a)
implica
moviendo(M, S, P, T_a)

R_2. Cuando uno selecciona el motor M y presiona el botón de posición BA, éste se empieza mover la carrera máxima del motor hacia la izquierda P_0.

ajustar motor(BA, M, T_a)
implica
moviendo(M, S = I, P = Pmáx, T_a)

R_3. Cuando la jeringa presiona el sensor de calibración SC el motor M se detiene.

ajustado(M, T_a)
implica
parado(M, BP = 0, T_a)

R_4. Cuando uno selecciona el motor M y presiona el botón de posición BPA, éste se mueve un paso.

Mover un paso(BPA, M, P, S, T)
implica
moviendo(M, S, P = 1, T_a)

R_5. Cuando se selecciona la posición del motor M situado a un número de pasos P y presiona el botón de calibrar, su carrera máxima toma el valor de éste número de pasos.

calibrar motor(BC, M, T)
implica
calibrado(M, P, BP = 5, T_a)

Reglas de control :

R_6. Cuando uno selecciona una posición P, el sistema calcula el número de pasos y selecciona la dirección (según la posición actual) para que el motor llegue a la posición deseada.Después le envía las instrucciones al microcontrolador.

mover motor (M, BP, T_a) y
posición(M, P, T, T_a)
implica
moviendo(M, P, S, T_1, T_2)

B) Especificación lógica del sistema de control del pulso

Identificadores :

R : Ritmo
BR : Botón de Ritmo
t_o : Es el tiempo inicial
T :Tiempo cuando ocurre el evento

Eventos:

- Envío de comandos al microcontrolador para que programe el temporizador con el ritmo deseado.

activar bomba(BR, R, T)
BR en [1..6], $T \geq t_0$

Estados:

- La bomba B se encuentra pulsando a un ritmo R.

pulsando(R, T_1, T_2)
R en [1..6] , $t_0 \leq T_1 < T_2$

Reglas de la bomba que relaciona los eventos con los estados :

R_1. Cuando se selecciona aceptar se cambia el ritmo.

activar bomba(BR, R, T_a)
implica
pulsando(R, T_a)

C) Especificación lógica del sistema de monitoreo

Identificadores:

M : Monitor
A/D : Convertidor analógico-digital
S : Señal desplegada
T_m : Tiempo de monitoreo
Ct : Cualquier tecla
t_0 : Es el tiempo inicial
T :Tiempo cuando ocurre el evento

Eventos :

- Envío de comandos a microcontrolador para activar el convertidor analógico-digital A/D que capture la señal del sensor S.

$$\text{aceptar}(A/D,\ T_m,\ T)$$
$$A/D \text{ en } [1..8],\ T_m,\ T \geq t_0$$

- Transcurre el tiempo de monitoreo.

$$\text{final}\ (A/D,\ T_m, T)$$
$$A/D \text{ en } [1..8],\ T_m,\ T \geq t_0$$

- Se presiona cualquier tecla.

$$\text{tecla}(Ct, T)$$
$$Ct \text{ en } [\text{true}, \text{false}],\ T \geq t_0$$

- Se cierra la ventana del monitor.

$$\text{salir}\ (\ )$$

Estados :

- El convertidor analógico digital esta enviando los datos al monitor M.

$$\text{capturando}(A/D,\ T_1,\ T_2)$$
$$A/D \text{ en } [1..8],\ t_0 \leq T_1 < T_2$$

- El monitor M esta recibiendo y desplegando en la pantalla los datos de la señal enviados por el microcontrolador.

$$\text{desplegando}(T_1,\ T_2)$$
$$t_0 \leq T_1 < T_2$$

- El último cuadro capturado se encuentra congelado.

congelado(S, T_1, T_2)
S en [1..8], $t_0 \leq T_1 < T_2$

Reglas del convertidor analógico digital que relaciona los eventos con los estados:

R_1. Al seleccionar el monitor de un sensor y presionar aceptar se envía los comandos pertinentes al microcontrolador para que empiece a digitalizar la señal y la envíe al la computadora.

aceptar(A/D, T_m, T_a)
implica
capturando(A/D, T_m, T_a) ***y***
desplegando(T_m, T_a)

R_2. Cuando se aprieta cualquier tecla y el monitor esta congelado, se cierra la ventana del monitor.

tecla(Ct, T)
y
congelado(S, T_1, T_2)
implica
salir ( )

Reglas de control :

R_3. Cuando transcurre el tiempo de monitoreo se para de muestrear y se congela el último cuadro.

final (A/D, T_m, T)
implica
congelado(S, T_a)

4.5 ANÁLISIS ORIENTADO A OBJETOS

4.5.1 Esquema del sistema

El esquema del sistema (Figura 4.1) permite a su diseñador concretizar los requerimientos en una representación gráfica, logrando una mayor comprensión del sistema. Como punto de partida se disponen todos los objetos que parecen necesarios para el diseño del problema. Luego se relacionan mediante flechas que indican las relaciones de uso de los objetos.

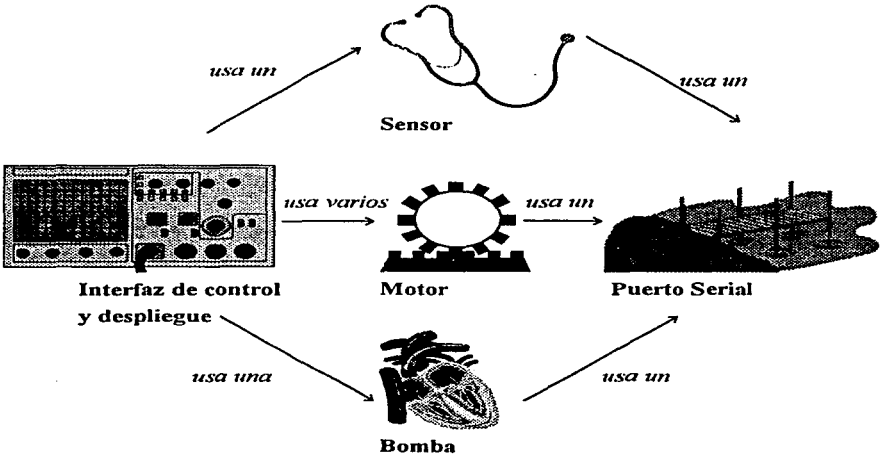


Figura 4.1 Dibujo de objetos con sus relaciones de uso.

Ya esbozada la interacción de las estructuras abstractas de datos que se necesita diseñar, se trata de delinear la funcionalidad que se debe incorporar mediante un diagrama a bloques.

4.5.2 Diagrama a bloques del sistema

El diagrama a bloques (Figura 4.2) descompone el programa en sistemas que interactuan entre si. De esta forma se puede apreciar como se tiene que modularizar las diferentes partes del código para que presenten un bajo acoplamiento y alta cohesión.

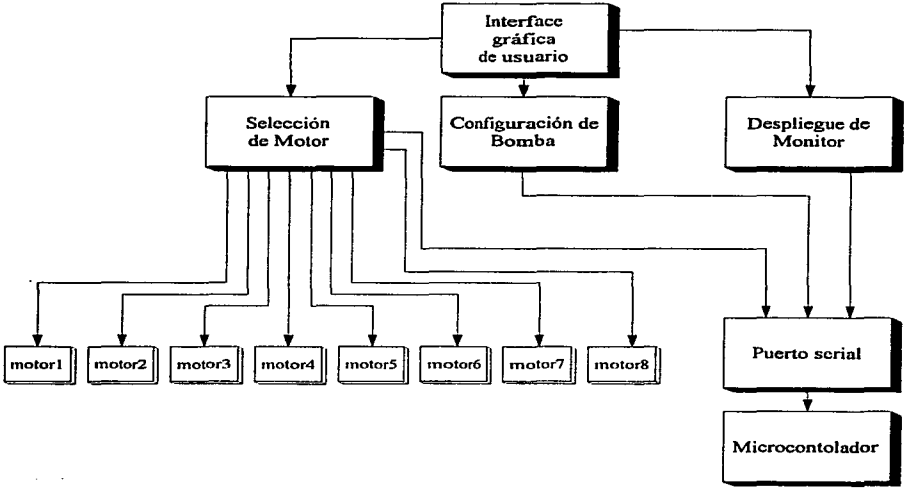


Figura 4.2 Diagrama a bloques del sistema (software).

En el diagrama bloques podemos observar que el usuario, a partir de la interface gráfica, opera los motores, la bomba y el monitor. Los sistemas *Selección de motor*, *configuración de bomba* y *despliegue del monitor* poseen una comunicación independiente con el puerto serial para obtener poco acoplamiento entre ellos de tal forma que se obtenga una buena modularidad.

4.6 Diseño orientado a objetos

Para el diseño orientado a objetos así como para la diagramación de las clases se seleccionó la metodología de Coad [9] . El método de diseño de Coad consiste en separar el sistema en :

- Componente de dominio del problema (PDC).
- Componente de interacción con el sistema (SIC).
- Componente de interacción con el humano (HIC)

Podemos identificar estos componentes gracias a las especificaciones anteriores donde se describe el propósito y las características de cada sistema. Podemos observar que el sistema de control de la oclusión tiene como propósito controlar la posición de los embolos.

4.6.1 Componente de Dominio del Problema (PDC)

Los requisitos del software del simulador del circulación cerebral en breve son :

- El control del ritmo del las pulsaciones.
- El control de la posición de los motores.
- El despliegue de la señal del un sensor de presión.

Por lo tanto necesitaremos tres clases iniciales que implementen cada una de las siguientes actividades el control de los motores, la configuración de la bomba y el despliegue del monitor:

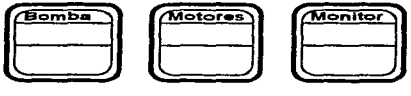


Figura 4.3 Clases del domino del problema.

4.6.1.1 Diseño de la clase *Bomba*

La tarea del objeto *Bomba* es proporcionar una cadena de caracteres en función de una llave con la cual se invoca su método *activaBomba*.

La entrada del método *ritmo* es un número del 1 al 5 que representa el ritmo al cual se desea que trabaje la bomba. Su salida será una cadena de caracteres que se almacenará en la dirección correspondiente del microcontrolador para que se programe el temporizador que controla la bomba:

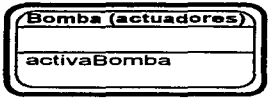


Figura 4.4 Clase *Bomba*

4.6.1.2 Diseño de las clases *Motores* y *Motor*

La clase *Motores* contiene los métodos que mandan las instrucciones al microcontrolador para que mueva, calibre o ajuste a 0 % el motor, y el atributo lista que contiene ocho objetos de Motor. En esta lista se encuentran los objetos *Motor* indexados y se pueden consultar o modificar fácilmente.

Se creó la clase *Motor* para manejar la estructura de datos tipo "Motor", mediante los atributos de *sentido*, *oclusión*, *identidad* y *pasos* .La clase *Motor* posee también el método de *CambiaOclusion*. Este método tiene como parámetro la posición a la cual se desea mover el motor y su labor es la de calcular cuantos pasos tiene que dar para alcanzar la posición seleccionada según su posición anterior.

Por lo tanto como entrada tiene una posición que es indicada mediante un número del 0 al 4 y que representa la oclusión de la arteria al 0%, 25%, 50%, 75% y 100%. Como salida almacena el sentido que tiene que tomar el motor y el número de pasos para llegar a la posición deseada en sus atributos *sentido* y *pasos* respectivamente.

Un diagrama de ambas clases se muestra a continuación :

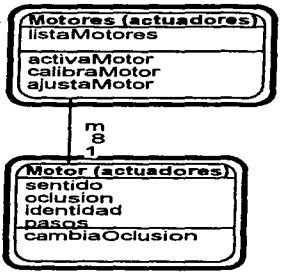


Figura 4.5 Clases *Motores* y *Motor*

4.6.1.3 Diseño de la clase *Monitor*

El monitor grafica los datos de uno de los sensores de presión durante un tiempo previamente configurado. Para este fin se creo el objeto *Monitor* con un método *grafica*. Este método se invoca con unos parámetros que le señalan cuanto tiempo se desea que se mantenga activo y cuál sensor se desea monitorear. Su entrada serán los datos que le proveerá el convertidor A/D del microcontrolador. Su salida será un oscilograma de presión (0 a 225 mm de Hg) contra tiempo (0 a 40 ms). Se creo el método *activaMonitor* para ejecutar la rutina en el microcontrolador que envíe los datos cuantizado del A/D a la computadora a través del puerta serial.

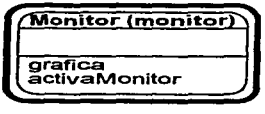


Figura 4.6 Clase *Monitor*

4.6.2 Componente de Interacción del Sistema (SIC)

Ya que se tiene diseñado el Componente de Domino del Problema, se puede diseñar aquel que le permitirá interactuar con otros sistemas : el Componente de Interacción del Sistema (SIC)

La función básica del software desarrollado es la de enviar al microcontrolador los comandos necesarios para que éste a su vez ejecute las rutinas para el control de los actuadores y la adquisición de datos. Esta tarea de comunicación entre el microcontrolador y la computadora se realiza por el puerto serial. Por lo tanto creó en el SIC (Figura 4.7) una clase llamada *Serial.* Este objeto se comportará como una interface entre el PDC y el microcontrolador.

Para una buena modularización se crearon varios objetos *Serial:* uno para el manejo de la *Bomba* , otro para los *Motores* y uno más para el *Monitor.* La clase *Bandera* nos permitirá llevar un correcto funcionamiento de las tres instancias. La forma en que realizará esta tarea se explicará posteriormente.

También se controlará el nivel de despliegue al usuario, gracias a su atributo *debug* que será heredado a las clases correspondientes para indicarles si debe de desplegar los mensajes de depuración del programa.

Figura 4.7 Clases de Interacción del Sistema

4.6.2.1 Diseño de la clase *Serial*

El objeto *Serial* (figura 4.8) tiene como tarea recibir y enviar los mensajes necesarios para el control de los dispositivos a través del microcontrolador. Para comunicarse a través del puerto serial se crearon dos métodos: un método de iniciación y otro de envío que contendrá el parámetro de instrucción.

El método *iniciacion* no tiene ningún parámetro. Su función será de vaciar el buffer de la computadora, inicializar el microcontrolador y configurar unas direcciones de memoria.

El método *envia*, recibe una cadena de caracteres y la manda al microcontrolador. Después recibe y despliega la respuesta del microcontrolador.

El método *CambiaponerStatus*, asigna al atributo *ponerStatus* un valor que indica si se necesita reinicializar el microcontrolador después de la ejecución del método *envia*.

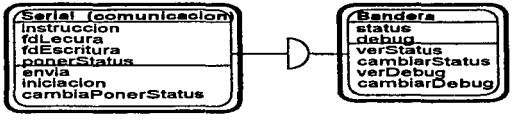


Figura 4.8 Clase *Serial*

4.6.2.2 Diseño de la clase *Bandera*

La clase Bandera realiza dos tareas. Contiene el atributo de clase para indicar al usuario cuando debe de reinicializar el microprocesador y nos dice si la aplicación se encuentra en un modo de depuración o un modo normal.

Para lograr una comunicación adecuada, las instancias de la clase *Serial* "comparten" el atributo *status* de la clase *Bandera*. Este atributo les indica que debe ser inicializado el microcontrolador siempre que la operación antes realizada haya sido la de monitorear un sensor, esto con el fin de detener el ciclo de conversión A/D. El atributo *status* se puede modificar o consultar desde los tres objetos *Serial* instanciados, gracias a los métodos *cambiarStatus* y *verStatus*. De esta forma las instancias de serial se pueden comunicar para enterarse en que tarea se encuentra el microcontrolador.

Para indicar a los objetos si desplieguan o no los mensajes de depuración se almacenó en una variable *debug* el número "0" , si nos encontramos en un modo normal, y el número 1 si nos encontramos en el modo de depuración (debug). Este atributo de clase puede ser accesado por medio de los métodos *cambiarDebug* y *verDebug*.

Figura 4.9 Clase *Bandera*

4.6.3 Componente de Interacción con el Humano (HIC)

Para el manejo del dominio del problema añadiremos el Componente de Interacción con el Humano (HIC), de tal forma que se pueda accesar de una manera amigable sus métodos y atributos.

Primero se diseño el método que crea la interface gráfica de usuario y después se programaron los métodos para accesar los objetos del PDC. El sistema cuenta con tres objetos que el usuario debe de manejar : los motores, la bomba y el monitoreo. Consecuentemente se creo una interface que esta dividida horizontalmente en tres partes. En la parte superior se encuentra el manejo de los motores, en la central se configura la bomba y en la parte inferior se activa el monitor. Una breve descripción de como el usuario interactuaria con la interface seria la siguiente:

Para el manejo de los motores, primero se selecciona que motor se desea mover, después se selecciona la posición donde queremos desplazarlo y finalmente se pulsa el botón "Activa motor".

Para la calibración motores, primero se selecciona el motor que se desea ajustar a 0% y se pulsa "Ajusta motor". Después se calibra el número de pasos para que el émbolo ocluya muestra arteria al 100% y se presiona "Motor Calibrado" para validar nuestra configuración.

Para la configuración de la bomba simplemente se selecciona el pulso deseado y se oprime " Activa bomba ".

Para el despliegue del monitor, primero seleccionamos el sensor que se quiere muestrear, después el tiempo que estará activo y finalmente se presiona "Activa monitor".

Si se quiere salir de la aplicación se pulsa el botón " Salir ".

En la figura 4.10 se muestra la forma en que la Interface Gráfica de Usuario aparece en pantalla.

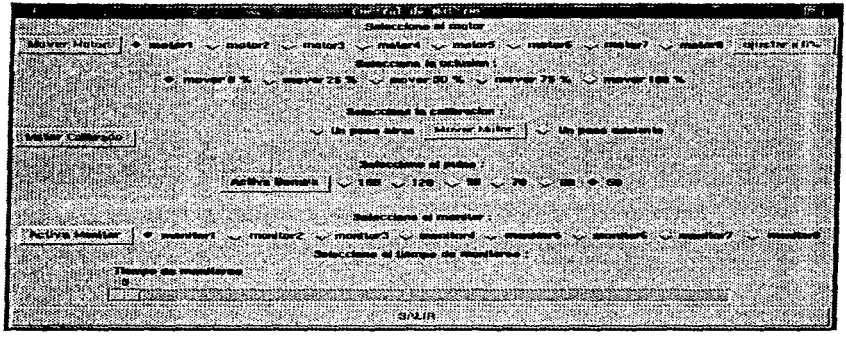


Figura 4.10 Interface Gráfica de Usuario

En la figura 4.11 se muestra el diagrama de la inteface gráfica. Ésta se realizó mediante las estructuras *todo-parte* que se dibujan como una línea con un triángulo. En nuestra aplicación la estructura *todo-parte* representa que el marco de la ventana *Gui* (marco principal) contiene a su vez tres marcos : el de motor, el de bomba y el de monitor. De igual forma estos marcos contienen instancias de botones, radiobotones y otros widgets.

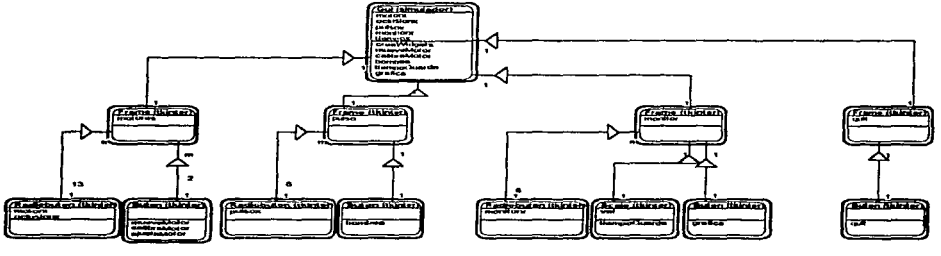


Figura 4.11 Diagrama de la HIC de acuerdo a la notación de Coad.

4.6.3.1.- Relación entre la Interface con el Humano, el Dominio del Problema y la Interacción del Sistema

En nuestro sistema existen cuatro tipos de relaciones : conexiones de objetos, mensajes, estructuras todo-parte y herencia.

Una conexión de objetos es simbolizada por una línea continua entre dos objetos y representa una asociación o un mapeo entre objetos. Por ejemplo, cada dispositivo tiene asociado un puerto serial para comunicarse con el microcontrolador.

Un mensaje es dibujado como una flecha que puede tener una etiqueta opcional y señala la interacción de un objeto a otro. Por ejemplo, cuando se aprieta el botón "Mover Motor" se invoca el método "mueveMotor".

La herencia se utilizó para comunicar a las diferentes instancias de los objetos *Serial* la última tarea que realizó el microcontrolador.

4.6.3.2 Conexiones entre Objetos

En este proyecto se conectan los objetos para tres propósitos :

- Manejar los objetos del PDC desde el HIC

  Dentro del constructor de la clase *Gui*, se instancian los tres objetos de dominio del problema. La interacción entre la HIC y el PDC se hace a través de la llamada de los métodos de los objetos instanciados en el constructor del HIC.

- Manejar del SIC desde el PDC

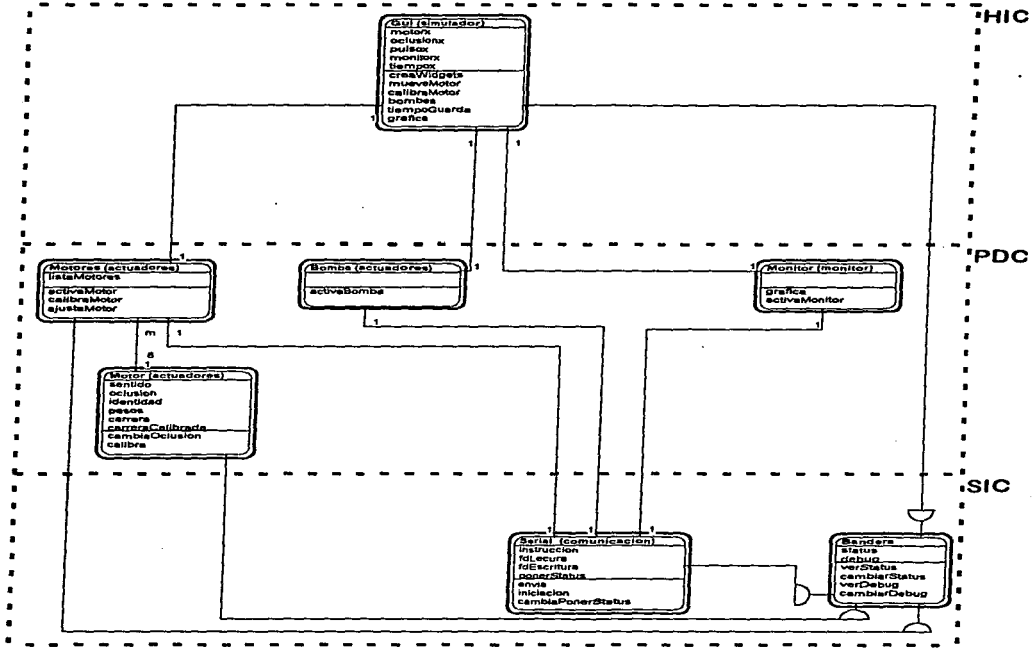
  Se puede observar en la figura 4.12 que cada objeto principal del PDC (*Motores*, *Bomba* y *Monitor*) tiene asociado un objeto Serial del SIC para comunicarse con el microcontrolador. Se asoció el objeto serial a cada objeto del PDC para que se puedan comunicar con el microcontrolador independientemente. Esto facilita en gran medida la reutilización y prueba de los objetos del PDC.

- Consultar o modificar la estructura de datos de los objetos *Motor*

  Dentro de una lista en la clase *Motores* se instancian ocho objetos *Motor* que pueden ser accesados a través de su posición en la lista o sea con el nombre de la lista y su índice.

Además existe una relación de herencia del SIC hacia el PDC y la HIC mediante la clase *Bandera*. Ésta última tiene dos propositos : Comunicar las instancias de *Serial* a través del atributo *status* y compartir la variable *debug* para indicar si el software debe de desplegar mensajes de depuración.

Figura 4.12 Relación entre el HIC, PDC y SIC



4.6.2.3 Mensajes entre Objetos

A continuación se presentan las cuatro interacciones de los mensajes del HIC con los objetos del PDC que constituyen parte del modelo dinámico del sistema orientado a objetos. Estas interacciones se muestran más claramente en la figura 4.13 .

1.- Para mover el motor:

- El usuario selecciona el motor a desplazar.
- El usuario escoge la posición deseada.
- El usuario presiona botón "Mover Motor" (mensaje A1).
- La computadora manda las instrucciones al microcontrolador para posicionar el motor gracias al método *mueveMotor* en la instancia de la clase *Motores*.

2.- Para ajustar el motor

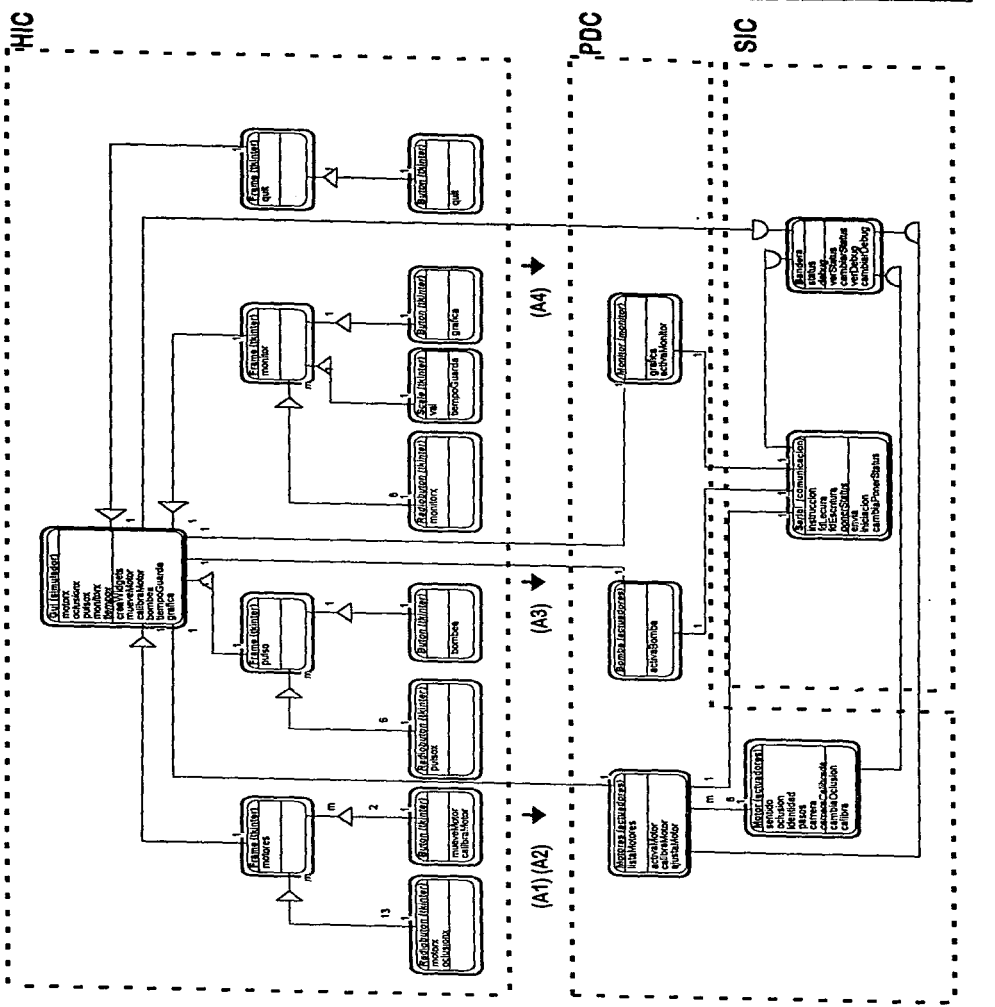
- El usuario selecciona el motor a calibrar.
- El usuario presiona botón "ajustar a 0%" (mensaje A2).
- La computadora manda las instrucciones al microcontrolador para calibrar el motor gracias al método *ajustaMotor* en la instancia de la clase *Motores*.

3.- Para configurar la bomba

- El usuario selecciona el ritmo que desea activar.
- El usuario presiona botón "Activa Bomba" (mensaje A3).
- La computadora manda las instrucciones al microcontrolador para activar la bomba por medio del método *activaBomba* en la instancia de la clase *Motores*.

4.- Para desplegar el monitor

- El usuario selecciona el monitor que desea desplegar.
- El usuario indica cuanto tiempo desea que se encuentre activo dicho monitor.
- El usuario presiona botón "Activa Monitor" (mensaje A4).
- La computadora manda las instrucciones al microcontrolador para configurar el convertidor A/D y que transmita los datos de la señal digitalizada.


HIC
PDC
SIC
(A1) (A2)
(A3)
(A4)

Conclusiones y comentarios

Debido a que el trabajo desarrollado se puede englobar dentro de un sistema típico de adquisición de datos y control de actuadores. El microcontrolador elegido fue el HC11, ya que este reúne las características necesarias para cumplir los requerimientos establecidos. Usar algún otro tipo de microcontrolador de más tecnología hubiera sido posible, sin embargo esto representaría una elevación en el costo del sistema.

Las actividades que el microcontrolador desempeña dentro del sistema son de comunicación serial con la computadora, manejo de variables dentro de su memoria interna, control de unidades lógicas mediante la escritura en sus puertos y digitalización de las señales analógicas. Esta última actividad es la que representa mayor desempeño y aunque el microcontrolador HC11 está limitado en cuestión de ancho de banda (1953.125 Hz en el modo utilizado en esta tesis), éste es suficiente para aplicaciones médicas, donde las señales no alcanzan más de 200 Hz.

En este sentido se realizaron pruebas de desempeño aplicando señales senoidales, cuadradas y triangulares al convertidor A/D del microcontrolador. Aunque la teoría marca un ancho de banda máximo de muestreo de 1953.125 Hz, dichas pruebas arrojaron un desempeño aceptable del sistema hasta los 1710 Hz. Decimos aceptable porque a esta frecuencia no se observa la modulación que la saturación del muestreo provocaría, sin embargo podemos notar distorsión en la señal muestreada.

Dicha distorsión no es perceptible a menos de 425 Hz .Esto se puede acreditar a que el sistema maneja una interpolación de primer orden al momento de su despliegue gráfico mediante la interface de usuario, así como a la interpolación que realiza el programa residente en el microcontrolador al tomar en cuenta sólo una de las cuatro conversiones realizadas en un mismo canal (425 Hz es aproximadamente la cuarta parte del ancho de banda práctico obtenido por las pruebas -1700 Hz-).

El empleo de un modulo de desarrollo de dicho microcontrolador nos da la ventaja de modularizar al sistema y hacerlo flexible a futuros cambios: si en algún momento se desea ampliar el ancho de banda del muestreo mediante el empleo de otro microcontrolador (HC16, por ejemplo), no es necesario cambiar la estructura física del resto de los módulos y bastaría con crear un link físico que permitiera acoplar el nuevo elemento al sistema.

De igual forma, cada uno de los módulos esta diseñado en forma flexible, permitiendo integrar nuevos elementos al sistema. Tal es el caso de la tarjeta maestra, en donde el bus de sensores maneja diversos niveles de voltaje, permitiendo así conectar a él casi cualquier tipo de transductor que el usuario determine. Otra ventaja en cuestión de hardware, además de la modularidad del sistema, es que la tecnología que complementa las funciones del microcontrolador es TTL lo que la hace estándar y de bajo costo.

El empleo de elementos de sistemas anteriores, así como de dispositivos que se podrían considerar como desperdicio (tales como los drives de computadora usados en el mecanismo de oclusión) disminuyen el costo del sistema y lo hacen susceptible a futuros cambios y mejoras. Este último aspecto es importante ya que, aunque el sistema fue hecho a la medida de los requerimientos solicitados, es susceptible a futuros cambios para su uso en cualquier aplicación en la que se requiera el monitoreo por computadora de alguna señal analógica y el control de actuadores. También existe la posibilidad de incrementar el desempeño del sistema ya que el microprocesador aún tiene recursos disponibles, tal como su puerto C (entrada y salida) que no fue utilizado en este sistema y puede ser aprovechado de la forma que nuevos usuarios determinen.

En caso de que nuevas investigaciones dentro del área determinen modificar los circuitos impresos ya diseñados, esto no representaría una gran complicación. El haber empleado una herramienta de alta tecnología en el diseño de esquemáticos e impresos (como lo es *Protel*) da la facilidad de editarlos en ambiente Windows, así como en cualquier otro paquete (tales como *OrCad* y *Tango*) ya que sus programas fuentes y sus bibliotecas son compatibles. Además da la oportunidad de simular el funcionamiento del sistema en paquetes tales como *Spice*.

Dentro del diseño del software, uno de los resultados más importantes en la tesis fue evaluar el desempeño de *Linux* y *Python* en una aplicación de instrumentación como una forma de desarrollo de sistemas con lenguajes alternativos. Esto sale de la línea comercial, dentro de la que encontramos a *Windows95* y lenguajes de programación como *C* o *C++*.

Se escogió desarrollar el programa en el sistema operativo *Linux*, dado que este provee muchos servicios y herramientas inexistentes en otros sistemas operativos de PC.
La flexibilidad y simplicidad en el manejo de dispositivos y archivos lo hacen un ambiente muy atractivo para el desarrollo de un aplicación de instrumentación. Sin embargo no se sabia como afectaría su naturaleza multitarea en situaciones de computo intensivo como el despliegue en tiempo real de los datos recibidos por el puerto serie. Finalmente se observó que el desempeño en graficación alcanzado era al deseado; resultó ser bastante veloz para

que no se llenara el buffer de entrada del puerto serial, y para que el usuario observe casi instantáneamente la señal en la ventana del monitor. Por lo tanto *Linux* resultó ser un sistema operativo adecuado para proveer instrumentación virtual en ciertas aplicaciones a pesar de ser multitarea.

Para desarrollar el programa se evaluaron varios lenguajes como *C*, *C++*, *Labview*, pero *Python* parecía más adecuado según su documentación. Sin embargo, no existía experiencia previa en el uso de este en una aplicación de instrumentación. Esto despertó la duda de si *Python* realmente ofrecía a los programadores la simplicidad y los beneficios de un corto ciclo de prueba que se buscaba en el lenguaje. Finalmente se observo que *Python* si cumplía los requerimientos que se esperaba. Por ello, podemos decir que *Python* representa una buena solución a la crisis de software actual donde la brecha entre el hardware y software aumenta cada vez más rápidamente. Gracias a la rapidez de procesamiento de las computadoras actuales, uno puede ahorrar tiempo de desarrollo y programación en un lenguaje de muy alto nivel como el *Python*, y sólo hacer programación de bajo nivel más laboriosa que consume más tiempo cuando sea necesario, para acoplarla a *Python*.

Anexo A : Hardware

A.1 Tarjeta maestra

A.1.1 Esquemáticos

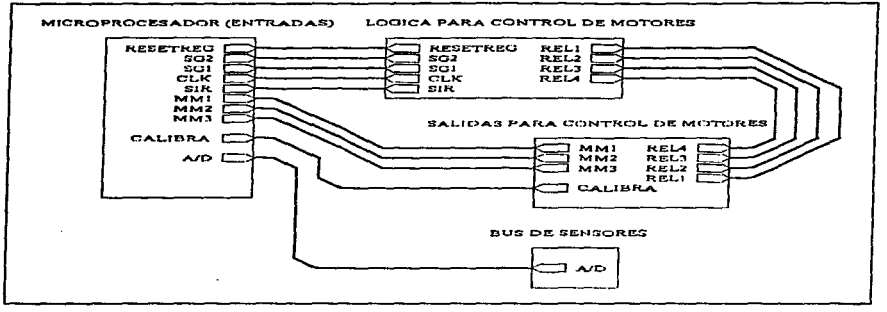


Figura A.1 Diagrama a bloques.

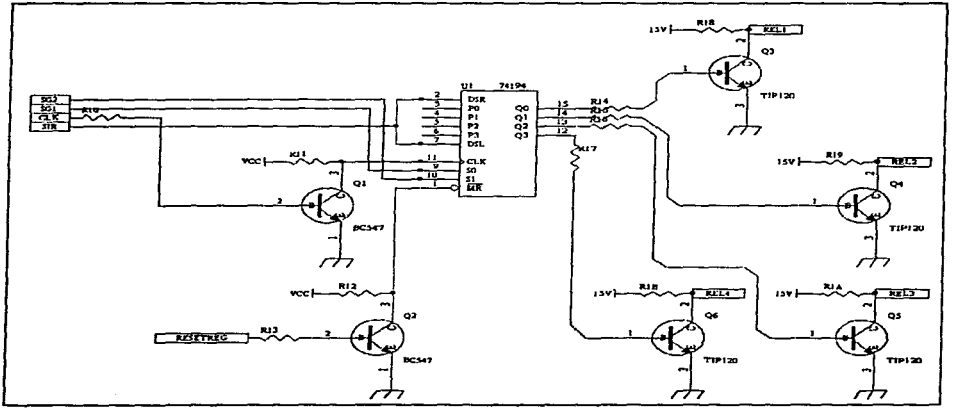


Figura A.2 Lógica para el control de motores.

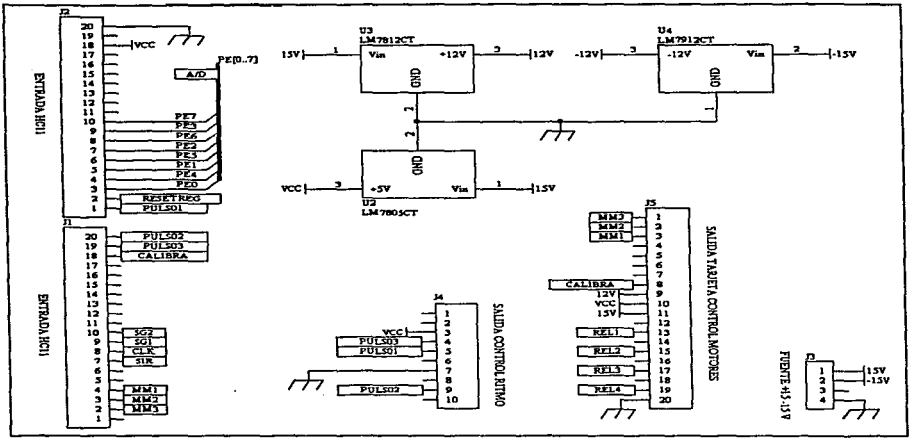


Figura A.3 Tarjeta maestra: Conectores de entrada y salida; alimentación.

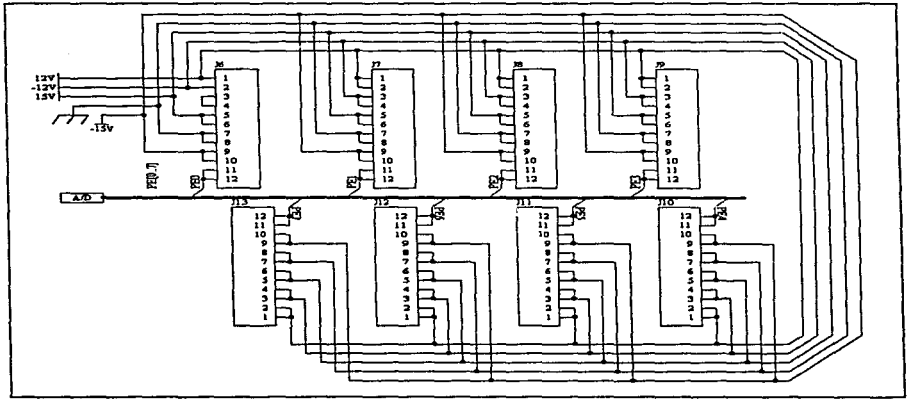
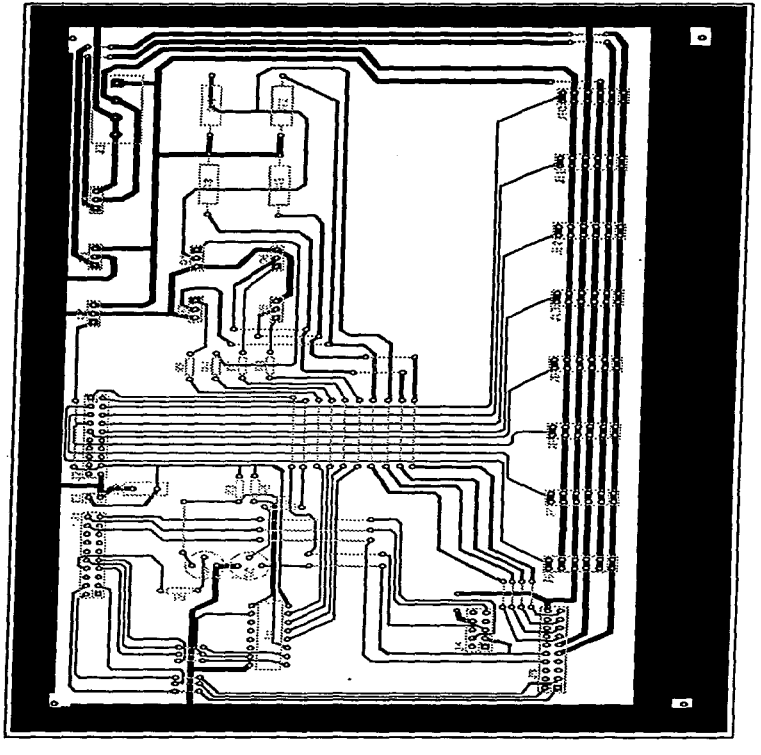


Figura A.4 Bus de sensores.

A.1.2 Circuito impreso



Circuito al 80 % del tamaño real

A.1.3 Lista de componentes

R1, R2	Resistencia 10 KΩ a 1/4 W
R3, R4	Resistencia 330 Ω a 1/4 W
R5, R6, R7, R8	Resistencia 150 KΩ a 1/4 W
R9, R10, R11, R12	Resistencia 680 Ω a 2 W
U1	74LS194
U2	LM7805
U3	LM7812
U4	LM7912
Q1, Q2	BC547
Q3, Q4, Q5, Q6	TIP120
C1	Capacitor de tantalio 1μF a 30 V
C2	Capacitor cerámico 0.01μF
J1, J2, J5	Conector IDC 20 pines
J3	Conector POWER 4
J4	Conector IDC 10 pines
J6, J7, J8, J9, J10, J11, J12, J13	Conector IDC 12 pines

A.2 Controlador de la bomba de flujo dicrótico

A.2.1 Esquemático

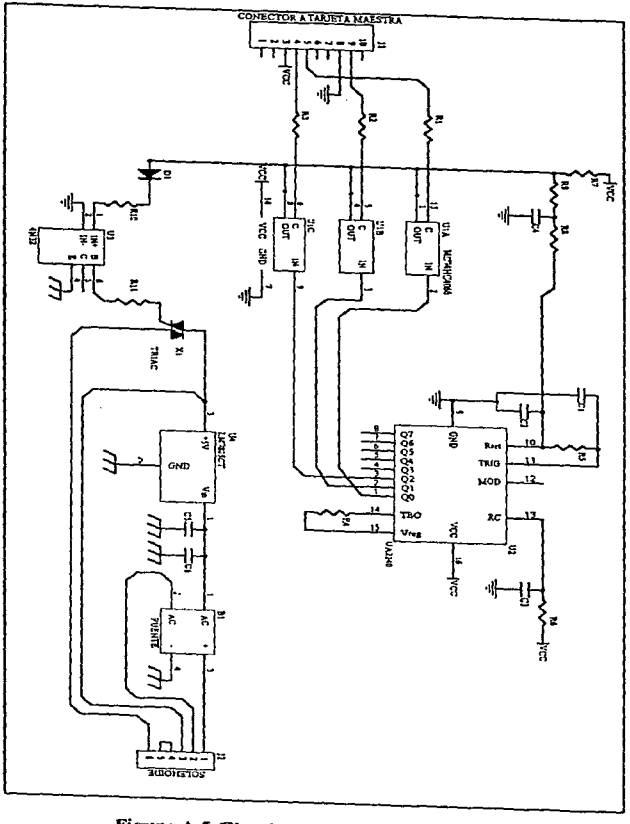
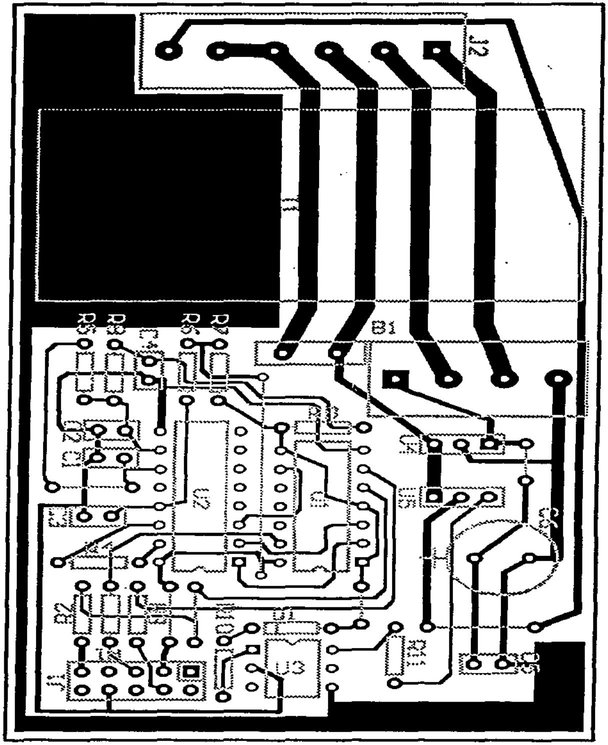


Figura A.5 Circuito de control de la bomba.

A.2.2 Circuito impreso



Circuito al 200% del tamaño real

A.2.3 Lista de componentes

R1, R2, R3	Resistencia 560 Ω a 1/4 W
R4	Resistencia 22 KΩ a 1/4 W
R5	Resistencia 5.1 KΩ a 1/4 W
R6	Resistencia 160 KΩ a 1/4 W
R7	Resistencia 3 KΩ a 1/4 W
R8, R9	Resistencia 5.6 KΩ a 1/4 W
R10	Resistencia 2.2 KΩ a 1/4 W
R11	Resistencia 22 KΩ a 1/4 W
U1	MC74HC4066
U2	UA2240
U3	4N32
U4	LM7805
U5	TIC206D
C1, C2, C3	Capacitor de poliester 0.01μF
C3, C5	Capacitor electrolítico 1μF
C6	Capacitor electrolítico 1000μF
D1	1N4001
T1	Transformador 110:12 a 250 mA
B1	RS203
J1	Conector IDC 10 pines
J2	Conector POWER 6

A.3 Acoplamiento de la señal del transductor

A.3.1 Esquemático

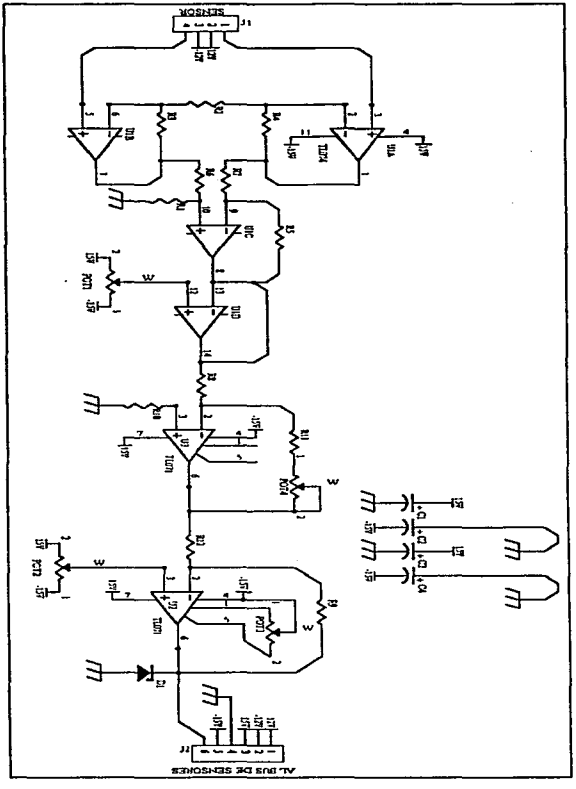
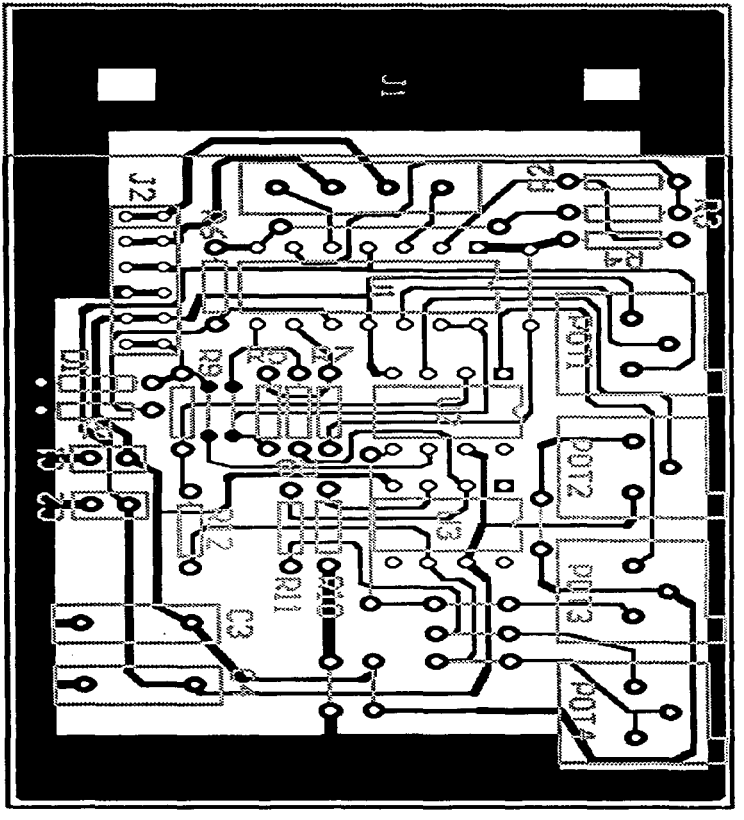


Figura A.6 Circuito de acondicionamiento de la señal.

A.3.2 Circuito impreso



Circuito al 240% del tamaño real

A.3.3 Lista de componentes

R1, R5, R9, R12	Resistencia 100 KΩ a 1/4 W
R2	Resistencia 2.2 KΩ a 1/4 W
R3, R4, R6, R7, R10	Resistencia 10 KΩ a 1/4 W
R8	Resistencia 1 KΩ a 1/4 W
R11	Resistencia 3.3 KΩ a 1/4 W
POT1, POT2, POT3	TRIGGERPOT 10 KΩ
POT4	TRIGGERPOT 1 KΩ
U1	TL074
U2, U3	TL071
D1	Zener 5.2 V
C1, C2	Capacitor de tantalio 1μF
C3, C4	Capacitor cerámico 0.01μF
J1	Conector Jack telefónico
J2	Conector IDC 6 pines

A.4 Tarjeta controladora de motores

A.4.1 Esquemáticos

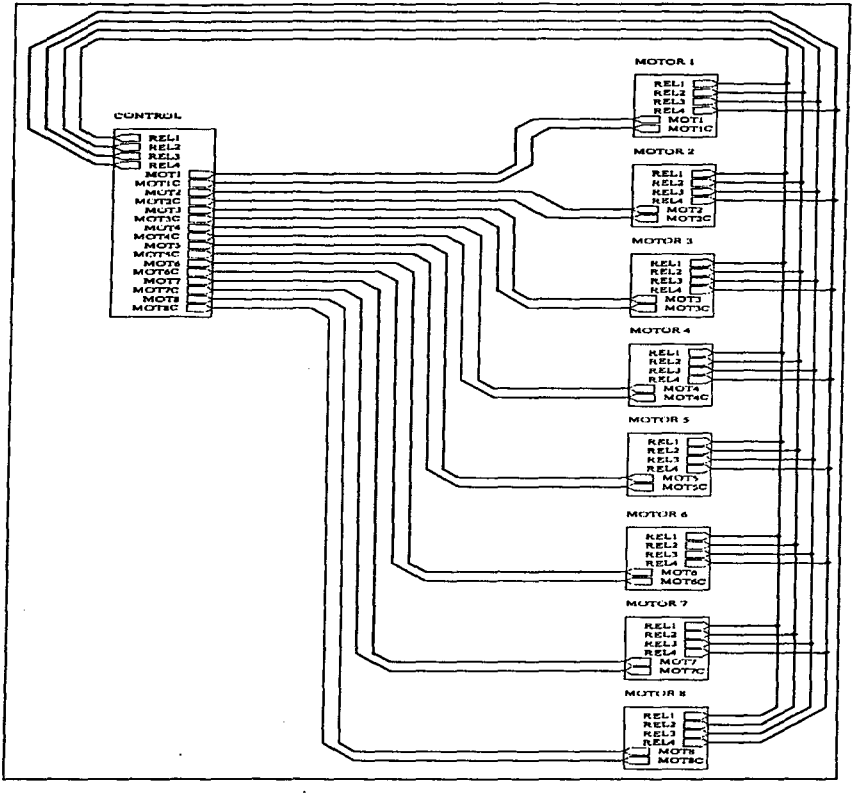


Figura A.7 Diagrama a bloques de la tarjeta.

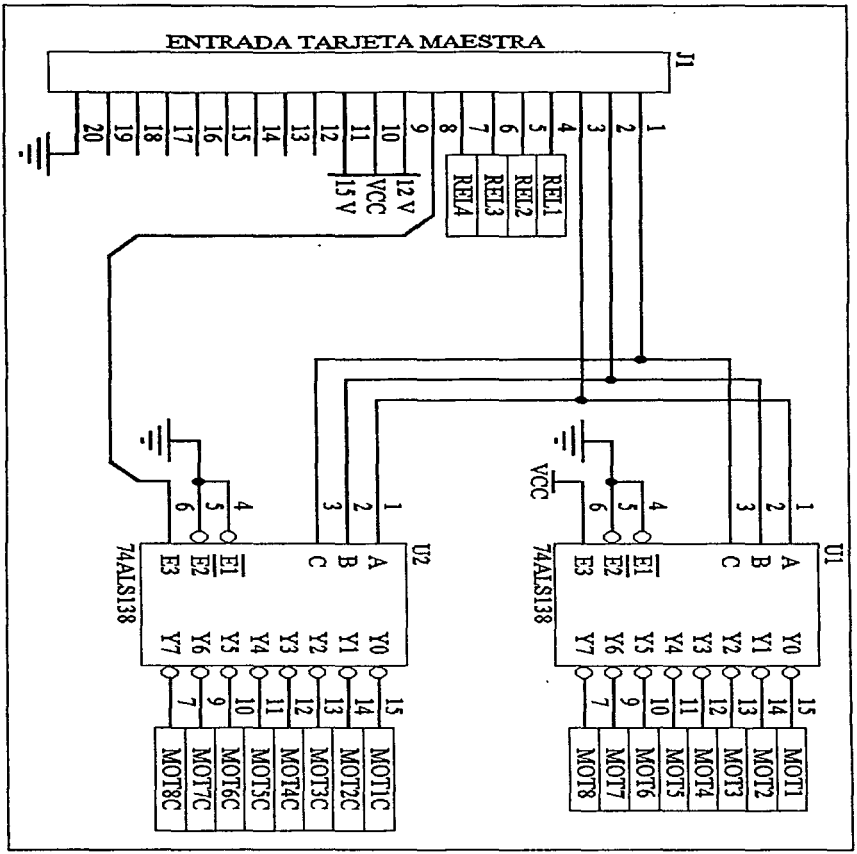


Figura A.8 Lógica de selección del motor y función de calibración.

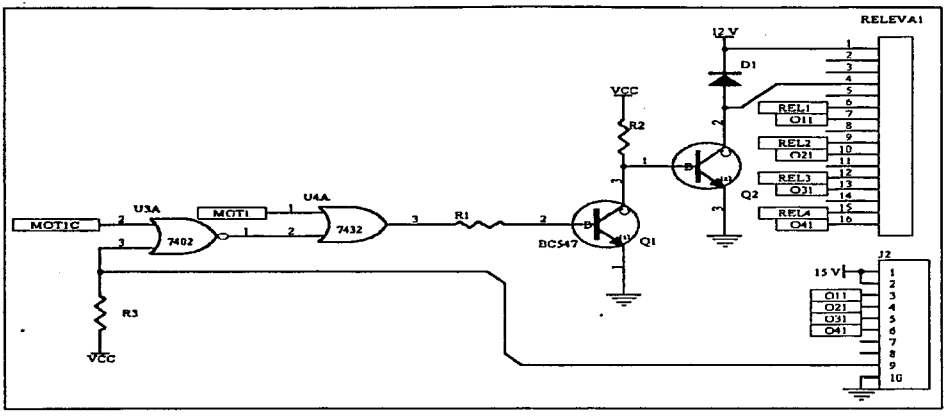


Figura A.9 Control del motor 1.

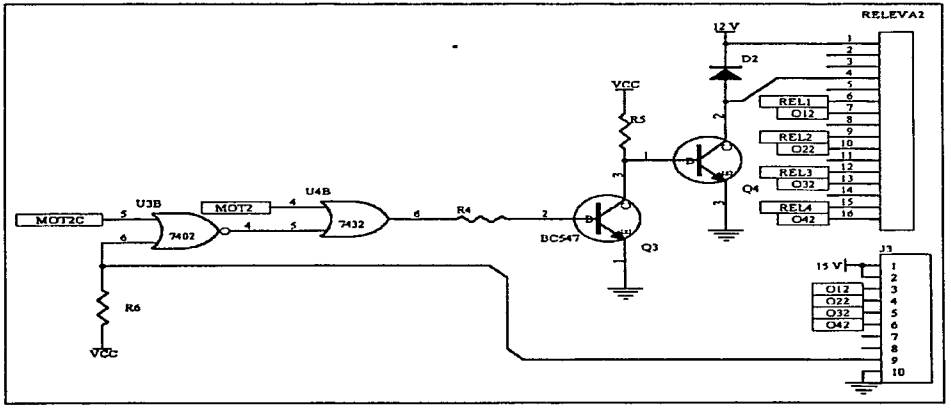


Figura A.10 Control del motor 2.

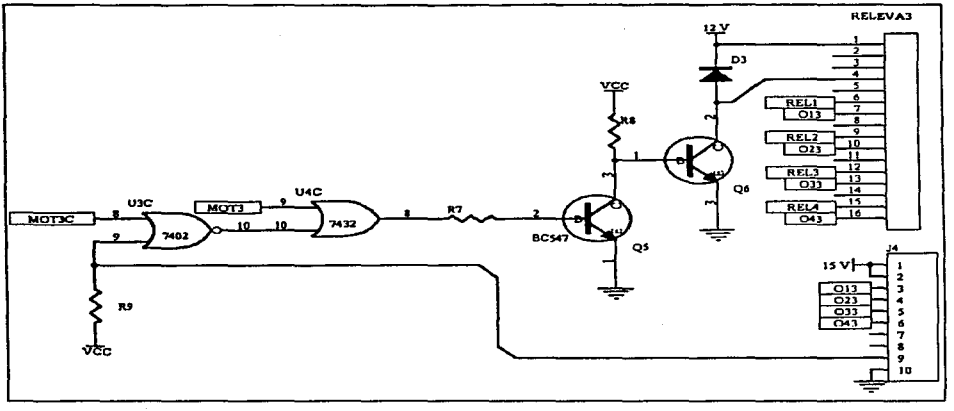


Figura A.11 Control del motor 3.

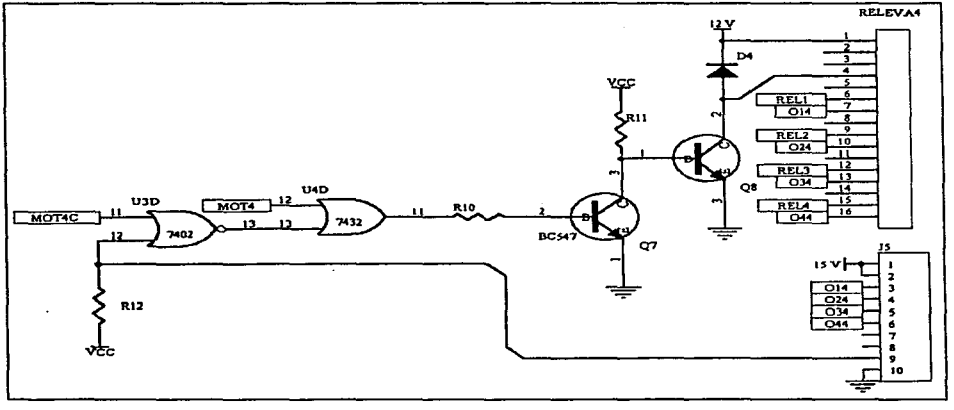


Figura A.12 Control del motor 4.

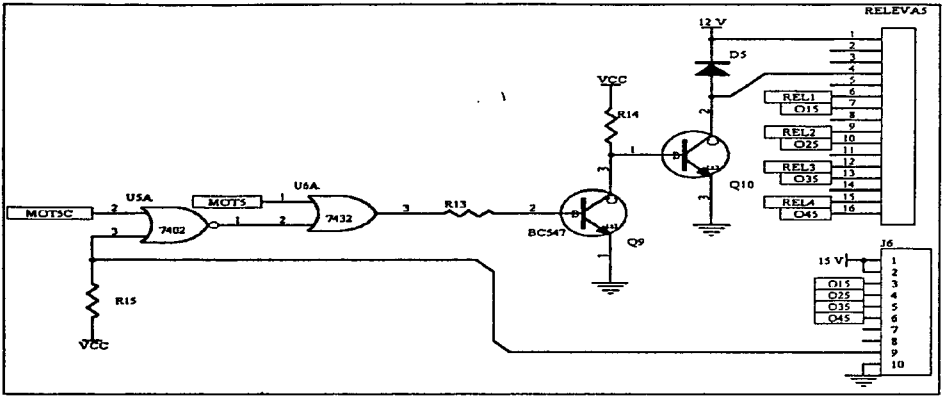


Figura A.13 Control del motor 5.

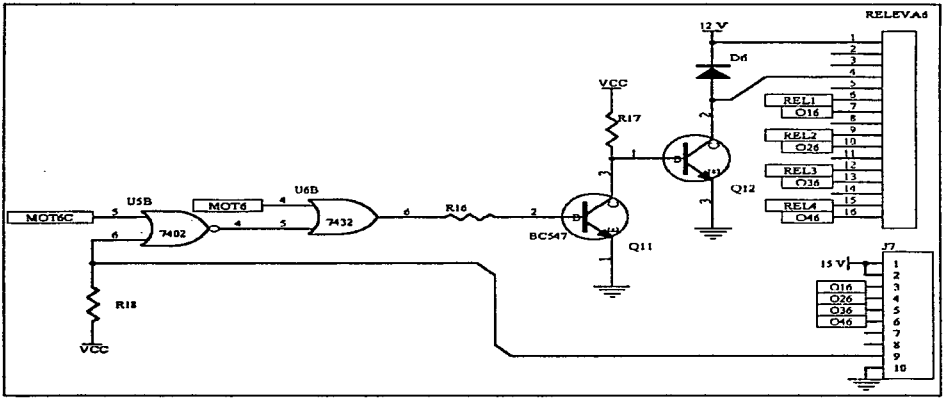


Figura A.14 Control del motor 6.

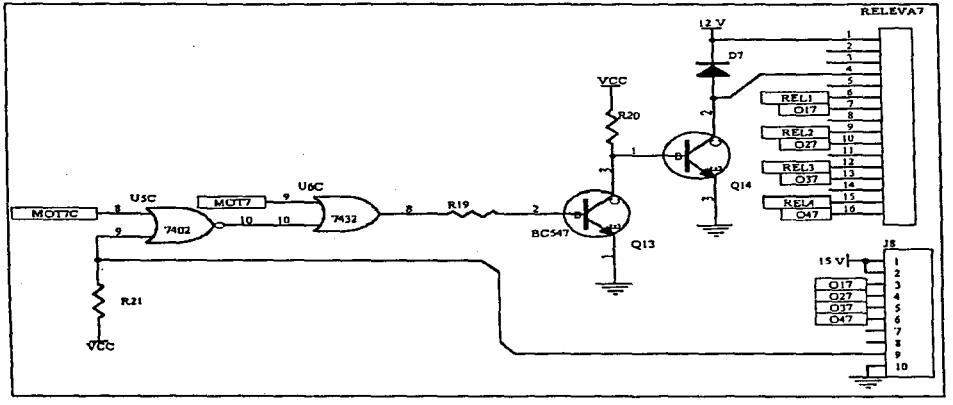


Figura A.15 Control del motor 7.

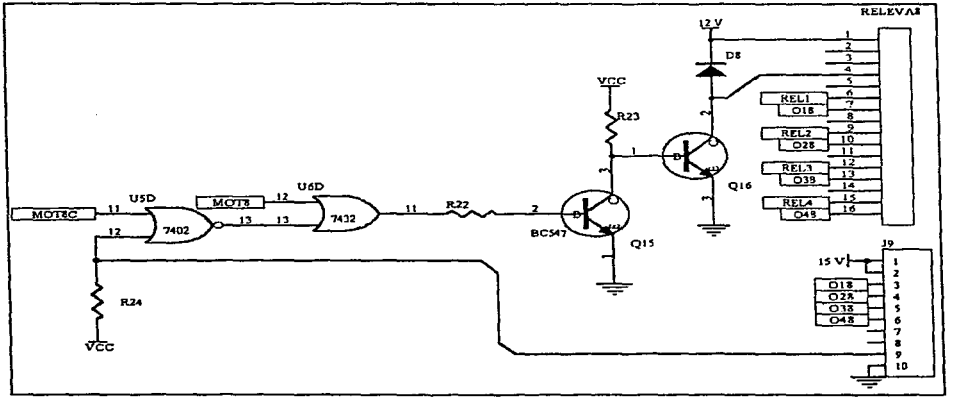
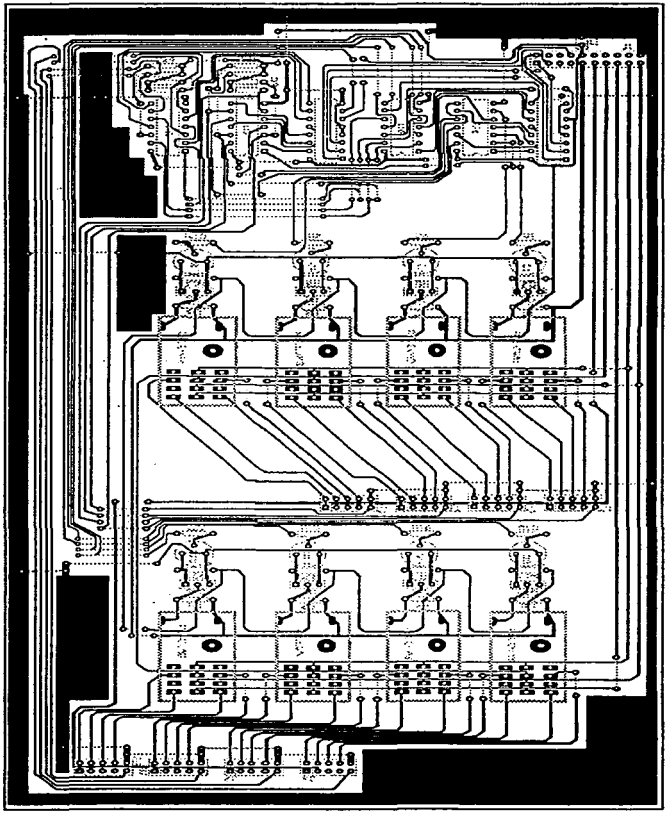


Figura A.16 Control del motor 8.

A.4.2 Circuito impreso



Circuito al 75 % del tamaño real

A.4.3 Lista de componentes

R1, R3, R4, R6, R7, R9, R10, R12, R13, R15, R16, R18, R19, R21, R22, R24	Resistencia 1 KΩ a 1/4 W
R2, R5, R8, R11, R14, R17, R20, R23	Resistencia 330 Ω a 1/4 W
U1, U2	74LS138
U3, U5	7402
U4, U6	7432
Q1, Q3, Q5, Q7, Q9, Q11, Q13, Q15	BC547
Q2, Q4, Q6, Q8, Q10, Q12, Q14, Q16	TIP120
D1, D2, D3, D4, D5, D6, D7, D8	IN4001
REL1, REL2, REL3, REL4, REL5, REL6, REL7, REL8	Relevador 4PDT 5A 12VDC
C1	Capacitor de tantalio 1μF a 30 V
J1	Conector IDC 20 pines
J2, J3, J4, J5, J6, J7, J8, J9	Conector IDC 10 pines

A.5 Hojas de especificaciones

A continuación se anexan las referencias en las cuales se pueden encontrar las hojas de especificaciones de los componentes electrónicos que fueron utilizados para el desarrollo del hardware* . Estas hojas incluyen el comportamiento del elemento bajo distintas condiciones, tablas de verdad (en el caso de componentes TTL) y aplicaciones generales, algunas de las cuales fueron utilizadas para el diseño de el sistema.

Componente	Referencia
XR-2240	EXAR Corporation. *DataBook: USIC's & ASIC's.*
4066B	Farchild . *CMOS DataBook.*
MC78xx	Motorola Semiconductor *Analog/Interface IC's Device Data Vol. 1.*
MC79xx	Motorola Semiconductor. *Analog/Interface IC's Device Data Vol. 1*
TL071C	Motorola Semiconductor *Analog/Interface IC's Device Data Vol. 1.*
TL074C	Motorola Semiconductor *Analog/Interface IC's Device Data Vol. 1.*
4N32	General Instruments. *Catalog of Optoelectronics Products.*
BC547C	Motorola Semiconductor. *Small Signal transistors, FET's and Diodes Device Data.*
TIP120	Motorola Semiconductor. *Bipolar Power Transistor Data.*
2N6073B (Equivalente al TIC206D)	Motorola Semiconductor. *Thyristor Device Data.*
74LS02	Motorola Semiconductor. *Fast and LS TTL Data.*

* Estas fueron las referencias utilizadas por los autores, más no significa que sean las únicas.

Componente	Referencia
74LS32	Motorola Semiconductor. *Fast and LS TTL Data.*
74LS138	Motorola Semiconductor. *Fast and LS TTL Data.*
74LS194	Motorola Semiconductor. *Fast and LS TTL Data.*

Anexo B : Software. Especificación de clases y relación entre objetos.

B.1 Clase *Bomba*

B.1.1 Especificación de la clase

Nombre de la clase :	Bomba
Módulo :	actuadores
Cardinalidad :	1
Implementación :	Python
Concurrencia :	Secuencial
Persistencia :	Transitoria
Tamaño :	Pequeño
Abstracto/Concreto :	Concreto
Indexada :	No
Atributos de Indexación :	N/A
Descripción :	Esta clase nos permite configurar la bomba.

B.1.2 Especificación de métodos

Nombre de la clase :	Bomba
Nombre de método :	activaBomba
Tipo de " Return " :	Nada
Nombre variable entrada :	pulso
Tipo de variable entrada :	String
Abstracto/Normal :	Normal
Categoría de la operación :	Selectora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Según la opción con la cual sea invocado el método envía una cadena de caracteres al microcontrolador para programar al temporizador.

B.2 Clase *Motores*

B.2.1 Especificación de la clase

Nombre de la clase :	Motores
Módulo :	actuadores
Cardinalidad :	1
Implementación :	Python
Concurrencia :	Secuencial
Persistencia :	Transitoria
Tamaño :	Pequeño
Abstracto/Concreto :	Concreto
Indexada :	No
Atributos de Indexación :	N/A
Descripción :	Esta clase crea un lista de instancias de clases indexadas *Motor* de tal forma que cada motor posee un atributo identidad, sentido, oclusión y pasos. También posee los métodos para enviar al microcontrolador las instrucciones para mover el motor.

B.2.2 Especificación de atributos

Nombre de la clase :	Motores
Nombre del atributo :	listaMotores
Tipo del atributo :	Lista de objetos de la clase Motor
Valor inicial :	00
Valor máximo :	07
Valor mínimo :	00
Tamaño :	Pequeño
Descripción :	Lista en el objeto *Motores* para accesar las instancias indexadas de la clase *Motor*

B.2.3 Especificación de métodos

Nombre de la clase :	Motores
Nombre de método :	activaMotor
Tipo de " Return " :	Nada
Nombre variable entrada :	opcion_motor , opcion_oclusion
Tipo de variable entrada :	Integer , Integer
Abstracto/Normal :	Normal
Categoría de la operación :	Selectora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Rutina de envío de la secuencia de instrucciones al microcontrolador para que mueva el motor seleccionado en la posición indicada

Nombre de la clase :	Motores
Nombre de método :	calibraMotor
Tipo de " Return " :	Motor
Nombre variable entrada :	opcion_motor
Tipo de variable entrada :	Integer
Abstracto/Normal :	Normal
Categoría de la operación :	selector
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Calibra la carrera del embolo al número de pasos necesario para producir una oclusión del 100% en la arteria.

Nombre de la clase :	Motores
Nombre de método :	ajustaMotor
Tipo de " Return " :	Nada
Nombre variable entrada :	opcion_motor
Tipo de variable entrada :	Integer
Abstracto/Normal :	Normal
Categoría de la operación :	Selectora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Regresa el embolo al principio de su carrera independientemente de la posición en que se encuentre.

B.3 Clase *Motor*

B.3.1 Especificación de la clase

Nombre de la clase :	Motor
Módulo :	actuadores
Cardinalidad :	m
Implementación :	Python
Concurrencia :	Secuencial
Persistencia :	Transitoria
Tamaño :	Pequeño
Abstracto/Concreto :	Concreto
Indexada :	Si
Atributos de Indexación :	Lista del 1 al 8
Descripción :	Esta clase sirve para crear objetos *Motor* de tal forma que cada motor posee un atributo de *identidad*, *sentido*, *oclusión* y *pasos*

B.3.2 Especificación de atributos

Nombre de la clase :	Motor
Nombre del atributo :	identidad
Tipo del atributo :	String
Valor inicial :	00
Valor máximo :	0C
Valor mínimo :	00
Tamaño :	Pequeño
Descripción :	Indica al microcontrolador que motor tiene que mover.

Nombre de la clase :	Motor
Nombre del atributo :	Sentido
Tipo del atributo :	String
Valor inicial :	"adelante"
Valor máximo :	"adelante"
Valor mínimo :	"atras"
Tamaño :	Pequeño
Descripción :	Indica al microcontrolador sentido en que debe girar el motor de pasos.

Nombre de la clase :	Motor
Nombre del atributo :	oclusion
Tipo del atributo :	Integer
Valor inicial :	0
Valor máximo :	5
Valor mínimo :	0
Tamaño :	Pequeño
Descripción :	Designa la posición del motor.

Nombre de la clase :	Motor
Nombre del atributo :	pasos
Tipo del atributo :	Integer
Valor inicial :	4
Valor máximo :	20
Valor mínimo :	4
Tamaño :	Pequeño
Descripción :	Es el número de pasos que se debe de mover el motor para llegar a la posición deseada.

Nombre de la clase :	Motor
Nombre del atributo :	carrera
Tipo del atributo :	Integer
Valor inicial :	0
Valor máximo :	20
Valor mínimo :	4
Tamaño :	Pequeño
Descripción :	Es el número de pasos al que se encuentra el motor cuando lo estamos calibrando para producir una oclusión del 100%.

Nombre de la clase :	Motor
Nombre del atributo :	carreraCalibrada
Tipo del atributo :	Integer
Valor inicial :	0
Valor máximo :	20
Valor mínimo :	4
Tamaño :	Pequeño
Descripción :	Es el número de pasos que produce una oclusión del 100%

B.3.3 Especificación de métodos

Nombre de la clase :	Motor
Nombre de método :	cambiaOclusion
Tipo de " Return " :	Nada
Nombre variable entrada :	posicion
Tipo de variable entrada :	Integer
Abstracto/Normal :	Normal
Categoría de la operación :	iterador
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Guarda la posición actual del motor, y según ésta y donde se desea mover, calcula el número de pasos que deben de darse.

Nombre de la clase :	Motor
Nombre de método :	calibra
Tipo de " Return " :	Nada
Nombre variable entrada :	N/A
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Modificadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Calibra la carrera del Motor, asignando a el atributo *carrera* el valor de *carreraCalibrada*.

B.4 Clase *Monitor*

B.4.1 Especificación de la clase

Nombre de la clase :	Monitor
Módulo :	monitor
Cardinalidad :	1
Implementación :	Python
Concurrencia :	Secuencial
Persistencia :	Transitoria
Tamaño :	Pequeño
Abstracto/Concreto :	Concreto
Indexada :	No
Atributos de Indexación :	N/A
Descripción :	Esta clase se encarga de graficar los datos que le entrega el convertidor A/D.

B.4.2 Especificación de métodos

Nombre de la clase :	Monitor
Nombre de método :	grafica
Tipo de " Return " :	Nada
Nombre variable entrada :	duracion
Tipo de variable entrada :	Integer
Abstracto/Normal :	Normal
Categoría de la operación :	Modificadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Grafica los datos que se reciben del puerto serial gracias al módulo de graficación científica Gist.

Nombre de la clase :	Monitor
Nombre de método :	activaMonitor
Tipo de " Return " :	Nada
Nombre variable entrada :	monitorx , duracion
Tipo de variable entrada :	String , Integer
Abstracto/Normal :	Normal
Categoría de la operación :	Selector
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Manda las instrucciones al microcontrolador para configurar el convertidor analógico-digital y empiece a mandar datos por el puerto serial.

B.5 Clase *Serial*

B.5.1 Especificación de clase

Nombre de la clase :	Serial
Módulo :	comunicacion
Cardinalidad :	3
Implementación :	Python
Concurrencia :	Secuencial
Persistencia :	Transitoria
Tamaño :	Pequeño
Abstracto/Concreto :	Concreto
Indexada :	No
Atributos de Indexación :	N/A
Descripción :	Esta clase nos permite interactuar con el microcontrolador por el puerto serial.

B.5.2 Especificación de atributos

Nombre de la clase :	Serial
Nombre del atributo :	fdEscritura
Tipo del atributo :	File Objet
Valor inicial :	N/A
Valor máximo :	N/A
Valor mínimo :	N/A
Tamaño :	Pequeño
Descripción :	File que se manda a través del puerto serial.

Nombre de la clase :	Serial
Nombre del atributo :	fdLectura
Tipo del atributo :	File Objet
Valor inicial :	N/A
Valor máximo :	N/A
Valor mínimo :	N/A
Tamaño :	Pequeño
Descripción :	File que se recibe a través del puerto serial.

Nombre de la clase :	Serial
Nombre del atributo :	instrucción
Tipo del atributo :	String
Valor inicial :	N/A
Valor máximo :	N/A
Valor mínimo :	N/A
Tamaño :	Pequeño
Descripción :	String que la instancia de *Serial* recibe para enviar mensaje al microcontrolador por medio de *FdEscritura*.

Nombre de la clase :	Serial
Nombre del atributo :	ponerStatus
Tipo del atributo :	Integer
Valor inicial :	N/A
Valor máximo :	N/A
Valor mínimo :	N/A
Tamaño :	Pequeño
Descripción :	Bandera que indica a un objeto serial si tiene que reinicializar el microcontrolador

B.5.3 Especificación de métodos

Nombre de la clase :	Serial
Nombre de método :	envia
Tipo de " Return " :	Nada
Nombre variable entrada :	instrucción
Tipo de variable entrada :	String
Abstracto/Normal :	Normal
Categoría de la operación :	Modificadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	N/A
Tamaño :	Pequeño
Descripción :	Envía string de instrucciones al microcontrolador.

Nombre de la clase :	Serial
Nombre de método :	iniciacion
Tipo de " Return " :	Nada
Nombre variable entrada :	Ninguna
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Modificadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	N/A
Tamaño :	Pequeño
Descripción :	Vacía el buffer del puerto serial de la computadora. Después envía un Line Feed al microcontrolador para ponerlo en línea e inicializa un rango de memoria. Luego de ésto, se le envían las instrucciones por medio del método *envia*.

Nombre de la clase :	Serial
Nombre de método :	cambiaPonerStatus
Tipo de " Return " :	Nada
Nombre variable entrada :	Var
Tipo de variable entrada :	Integer
Abstracto/Normal :	Normal
Categoría de la operación :	Modificadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	Pequeño
Tamaño :	Pequeño
Descripción :	Asigna al atributo *ponerStatus* un valor que indica si es necesario reinicializar el microcontrolador despues de la ejecución del método *envia*.

Nombre de la clase :	Serial
Nombre de método :	abre_reser
Tipo de " Return " :	
Nombre variable entrada :	
Tipo de variable entrada :	Integer
Abstracto/Normal :	Normal
Categoría de la operación :	Modificadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	Pequeño
Tamaño :	Pequeño
Descripción :	Asigna al atributo *ponerStatus* un valor que indica si es necesario reinicializar el microcontrolador despues de la ejecución del método *envia*.

B.6 Clase *Bandera*

B.6.1 Especificación de la clase

Nombre de la clase :	Bandera
Módulo :	bandera
Cardinalidad :	1
Implementación :	Python
Concurrencia :	Secuencial
Persistencia :	Transitoria
Tamaño :	Pequeño
Abstracto/Concreto :	Abstracto
Indexada :	No
Atributos de Indexación :	N/A
Descripción :	Esta clase nos permite sincronizar los objetos *Serial* y cambiar de un modo de ejecución normal a uno de depuración.

B.6.2 Especificación de atributos

Nombre de la clase :	Bandera
Nombre del atributo :	status
Tipo del atributo :	Integer "Booleano"
Valor inicial :	0
Valor máximo :	1
Valor mínimo :	0
Tamaño :	Pequeño
Descripción :	Atributo de clase que indica a los objetos *Serial* si se tiene que reinicializar el microcontrolador.

Nombre de la clase :	Bandera
Nombre del atributo :	debug
Tipo del atributo :	Integer "Booleano"
Valor inicial :	0
Valor máximo :	1
Valor mínimo :	0
Tamaño :	Pequeño
Descripción :	Atributo de clase para indicar a los objetos que lo hereden si deben de desplegar los mensajes para depuración del sistema.

B.6.3 Especificación de métodos

Nombre de la clase :	Bandera
Nombre de método :	cambiarStatus
Tipo de " Return " :	N/A
Nombre variable entrada :	var
Tipo de variable entrada :	Integer
Abstracto/Normal :	Abstracta
Categoría de la operación :	Modificador
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	N/A
Tamaño :	Pequeño
Descripción :	Guarda en la variable de clase *status* el parametero entero 0 ó 1 con el cual es invocado.

Nombre de la clase :	Bandera
Nombre de método :	verStatus
Tipo de " Return " :	Integer
Nombre variable entrada :	N/A
Tipo de variable entrada :	N/A
Abstracto/Normal :	Abstracto
Categoría de la operación :	selector
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	N/A
Tamaño :	Pequeño
Descripción :	Devuelve el entero 0 ó 1 de la variable de clase *status*.

Nombre de la clase :	Bandera
Nombre de método :	cambiarDebug
Tipo de " Return " :	Nada
Nombre variable entrada :	N/A
Tipo de variable entrada :	N/A
Abstracto/Normal :	Abstracto
Categoría de la operación :	Modificador
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	N/A
Tamaño :	Pequeño
Descripción :	Guarda en la variable de clase *debug* el parametero entero 1.

Nombre de la clase :	Bandera
Nombre de método :	verDebug
Tipo de " Return " :	Integer
Nombre variable entrada :	N/A
Tipo de variable entrada :	N/A
Abstracto/Normal :	Abstracto
Categoría de la operación :	Selector
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	N/A
Tamaño :	Pequeño
Descripción :	Devuelve el entero 0 ó 1 de la variable de clase *debug*.

B.7 Clase *Gui*

B.7.1 Especificación de la clase

Nombre de la clase :	Gui
Módulo :	scc
Cardinalidad :	1
Implementación :	Python
Concurrencia :	Secuencial
Persistencia :	Transitoria
Tamaño :	Pequeño
Abstracto/Concreto :	Concreto
Indexada :	No
Atributos de Indexación :	N/A
Descripción :	Esta clase sirve para crear la interface gráfica con los Widgets de TCL/TK.

B.7.2 Especificación de atributos

Nombre de la clase :	Gui
Nombre del atributo :	pulsox
Tipo del atributo :	Integer
Valor inicial :	"00"
Valor máximo :	"04"
Valor mínimo :	"00"
Tamaño :	Pequeño
Descripción :	Pulso que tendrá la bomba de flujo diastólico.

Nombre de la clase :	Gui
Nombre del atributo :	motorx
Tipo del atributo :	Integer
Valor inicial :	"00"
Valor máximo :	"07"
Valor mínimo :	"00"
Tamaño :	Pequeño
Descripción :	Número con el cuál se accesa la lista en la clase *Motores* para crear las instancias indexadas de la clase *Motor*.

Nombre de la clase :	Gui
Nombre del atributo :	oclusionx
Tipo del atributo :	Integer
Valor inicial :	"00"
Valor máximo :	"11"
Valor mínimo :	"00"
Tamaño :	Pequeño
Descripción :	Mensaje que se envia al microcontrolador para que llegue a la posición deseada.

Nombre de la clase :	Gui
Nombre del atributo :	monitorx
Tipo del atributo :	String
Valor inicial :	0
Valor máximo :	7
Valor mínimo :	0
Tamaño :	Pequeño
Descripción :	Canal del convertidor A/D del microcontrolador seleccionado para ser desplegado.

Nombre de la clase :	Gui
Nombre del atributo :	tiempox
Tipo del atributo :	Integer
Valor inicial :	5
Valor máximo :	420
Valor mínimo :	5
Tamaño :	Pequeño
Descripción :	Tiempo en segundos que durará activo el monitor.

B.7.3 Especificación de métodos

Nombre de la clase :	Gui
Nombre de método :	creaWidgets
Tipo de " Return " :	Nada
Nombre variable entrada :	Nada
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	constructor
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Instancia la interface gráfica.

Nombre de la clase :	Gui
Nombre de método :	mueveMotor
Tipo de " Return " :	Nada
Nombre variable entrada :	Nada
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Seleccionadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Manda las instrucciones correspondientes al microcontrolador para posicionar el motor en el lugar seleccionado.

Nombre de la clase :	Gui
Nombre de método :	calibraMotor
Tipo de " Return " :	Nada
Nombre variable entrada :	Nada
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Seleccionadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Manda las instrucciones para calibrar a una oclusión de 100% según el número de pasos que se movió el motor para llegar a la posición actual.

Nombre de la clase :	Gui
Nombre de método :	ajustaMotor
Tipo de " Return " :	Nada
Nombre variable entrada :	Nada
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Seleccionadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Manda las instrucciones correspondientes al microcontrolador para ajustar el motor en la posición cero.

Nombre de la clase :	Gui
Nombre de método :	bombea
Tipo de " Return " :	Nada
Nombre variable entrada :	Nada
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Seleccionadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Manda las instrucciones correspondientes al microcontrolador para configurar la bomba al ritmo deseado.

Nombre de la clase :	Gui
Nombre de método :	tiempoGuarda
Tipo de " Return " :	Val
Nombre variable entrada :	Integer
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Modificadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Guarda en el atributo *tiempox* el valor de la variable *val* del widget scale.

Nombre de la clase :	Gui
Nombre de método :	grafica
Tipo de " Return " :	Nada
Nombre variable entrada :	Nada
Tipo de variable entrada :	N/A
Abstracto/Normal :	Normal
Categoría de la operación :	Seleccionadora
Implementación :	Python
Transformación :	N/A
Nombre de excepción :	Ninguna
Tipo de excepción :	N/A
Invariante Aplicable :	Ninguna
Concurrencia :	Secuencial
Tiempo de ejecución :	pequeño
Tamaño :	Pequeño
Descripción :	Manda las instrucciones correspondientes al microcontrolador para monitorear el sensor de presión indicado. Activa durante un tiempo preestablecido, mediante el widget scale, la graficación en un ventana y congela el último cuadro.

B.8 Especificación de relaciones entre objetos

Clase fuente :	Gui
Clase destino :	Motores
Atributo clase/tipo transversal y nombre :	Motores M
Orden clase/tipo transversal y nombre :	N/A
Tipo de relación :	Agregación
Cardinalidad de la relación :	1 a 1
Atributo inverso, transversal clase/tipo y nombre :	Gui app
Implementación :	Python

Clase fuente :	Gui
Clase destino :	Bomba
Atributo clase/tipo transversal y nombre :	Bomba B
Orden clase/tipo transversal y nombre :	N/A
Tipo de relación :	Agregación
Cardinalidad de la relación :	1 a 1
Atributo inverso, transversal clase/tipo y nombre :	Gui app
Implementación :	Python

Clase fuente :	Gui
Clase destino :	Monitor
Atributo clase/tipo transversal y nombre :	Monitor G
Orden clase/tipo transversal y nombre :	N/A
Tipo de relación :	Agregación
Cardinalidad de la relación :	1 a 1
Atributo inverso, transversal clase/tipo y nombre :	Gui app
Implementación :	Python con Numeric y Gist

Clase fuente :	Motores
Clase destino :	Serial
Atributo clase/tipo transversal y nombre :	Serial C
Orden clase/tipo transversal y nombre :	N/A
Tipo de relación :	Agregación
Cardinalidad de la relación :	1 a 1
Atributo inverso, transversal clase/tipo y nombre :	Monitor G
Implementación :	Python con utilerías Posix del RedHat

Clase fuente :	Monitor
Clase destino :	Serial
Atributo clase/tipo transversal y nombre :	Serial C
Orden clase/tipo transversal y nombre :	N/A
Tipo de relación :	Agregación
Cardinalidad de la relación :	1 a 1
Atributo inverso, transversal clase/tipo y nombre :	Monitor G
Implementación :	Python con utilerías Posix del RedHat

Clase fuente :	Bomba
Clase destino :	Serial
Atributo clase/tipo transversal y nombre :	Serial C
Orden clase/tipo transversal y nombre :	N/A
Tipo de relación :	Agregación
Cardinalidad de la relación :	1 a 1
Atributo inverso, transversal clase/tipo y nombre :	Monitor G
Implementación :	Python con utilerías Posix del RedHat

Clase fuente :	Motores
Clase destino :	Motor
Atributo clase/tipo transversal y nombre :	Motor Motores[i]
Orden clase/tipo transversal y nombre :	N/A
Tipo de relación :	Agregación
Cardinalidad de la relación :	1 a 1
Atributo inverso, transversal clase/tipo y nombre :	Monitor G
Implementación :	Python

Anexo C : Software. Código fuente.

```
# Nombre de modulo:
# Clases dentro del modulo: 1
#                               - Gui


# Importacion de modulos necesarios

import sys              # Se necesita para capturar el argumento de iniciacion debug
from Tkinter import *   # Se necesita para tener acceso a un gui con widgets
import string           # Se necesita para cambiar de tipo string a int
import actuadores       # Con este se instaciara motor para coveritr la opcion n al
                        # codigo del micro
import comunicacion     # Se necesita para comunicarse al puerto
import monitor          # Es el modulo que contiene los monitores para visualizar
                        # las las senales en
                        # los canales del convertidor A/D del microcontrolador
import bandera          # Sirve para heredar el atributo de clase de control de debug


# Clase Gui, crea la interface grafica de usuario con widgets de TCL/TK a traves
# de modulo Tkinter.

class Gui(Frame,bandera.Bandera):

    # Metodo creaWidgets,Inicializacion y instanciazion de los widgets
    # Instancia.creaWidgets()

    def creaWidgets(self):

        #
        # Declaracion de literales "default"
        #

        self.motorx     = IntVar()
        self.motorx.set(0)
        self.oclusionx = IntVar()
        self.oclusionx.set(0)
        self.pulsox     = IntVar()
        self.pulsox.set(0)
        self.monitorx   = StringVar()
        self.monitorx.set(0)
        self.tiempox    = IntVar()
        self.tiempox.set(0)

        #
        # Creacion de Frame principal con el titulo " Control de Motores "
        #

        self.top = self.master
        # EMPIEZA EL FRAME TOP
        self.top.title(" Control de Motores ")

        #
        # Titulo del Frame motores con el titulo     " Selecione el motor "
        #

        self.motores = Frame(self.top)
        # EMPIEZA EL FRAME MOTORES
        self.motores.label = Message(self.motores,
                                { "width" : "5i",
```

```
                        "text" : " Selecione el motor "})

    self.motores.label.pack(fill=X)

    #
    # Boton para uso del objeto motor, invoacion del metodo mueveMotor
    #

    self.motores.motor = Frame(self.motores)
    # EMPIEZA EL FRAME MOTORES.MOTOR
    self.motores.motor.boton = Button(self.motores.motor,
                              {'text': 'Mover Motor',
                               'fg': 'blue',
                               'command' : self.mueveMotor})

    self.motores.motor.boton.pack({"side":"left"})

    #
    # Radiobotones que selecionan que motor deseamos mover
    #

    self.motores.motor.motor1 = Radiobutton (self.motores.motor,
                                  {"text" : "motor1",
                                   "variable" : self.motorx,
                                   "value" : 0,
                                   "anchor" : "w",
                                   Pack : {"side" : "top",
                                           "fill" : "x"}
                                   }
                                  )

    self.motores.motor.motor1.pack({"side":"left"})
    self.motores.motor.motor2 = Radiobutton (self.motores.motor,
                                  {"text" : "motor2",
                                   "variable" : self.motorx,
                                   "value" : 1,
                                   "anchor" : "w",
                                   Pack : {"side" : "top",
                                           "fill" : "x"}
                                   }
                                  )
    self.motores.motor.motor2.pack({"side":"left"})
    self.motores.motor.motor3 = Radiobutton (self.motores.motor,
                                  {"text" : "motor3",
                                   "variable" : self.motorx,
                                   "value" : 2,
                                   "anchor" : "w",
                                   Pack : {"side" : "top",
                                           "fill" : "x"}
                                   }
                                  )
    self.motores.motor.motor3.pack({"side":"left"})
    self.motores.motor.motor4 = Radiobutton (self.motores.motor,
                                  {"text" : "motor4",
                                   "variable" : self.motorx,
                                   "value" : 3,
                                   "anchor" : "w",
                                   Pack : {"side" : "top",
                                           "fill" : "x"}
                                   }
```

```
                                    )
 self.motores.motor.motor4.pack({"side":"left"})
 self.motores.motor.motor5 = Radiobutton (self.motores.motor,
                                    {"text" : "motor5",
                                     "variable" : self.motorx,
                                     "value" : 4,
                                     "anchor" : "w",
                                     Pack : {"side" : "top",
                                             "fill" : "x"}
                                     }
                                    )
 self.motores.motor.motor5.pack({"side":"left"})
 self.motores.motor.motor6 = Radiobutton (self.motores.motor,
                                    {"text" : "motor6",
                                     "variable" : self.motorx,
                                     "value" : 5,
                                     "anchor" : "w",
                                       Pack : {"side" : "top",
                                               "fill" : "x"}
                                     }
                                    )
 self.motores.motor.motor6.pack({"side":"left"})
 self.motores.motor.motor7 = Radiobutton (self.motores.motor,
                                    {"text" : "motor7",
                                     "variable" : self.motorx,
                                     "value" : 6,
                                     "anchor" : "w",
                                     Pack : {"side" : "top",
                                             "fill" : "x"}
                                     }
                                    )
 self.motores.motor.motor7.pack({"side":"left"})
 self.motores.motor.motor8 = Radiobutton (self.motores.motor,
                                    {"text" : "motor8",
                                     "variable" : self.motorx,
                                     "value" : 7,
                                     "anchor" : "w",
                                     Pack : {"side" : "top",
                                             "fill" : "x"}
                                     }
                                    )
 self.motores.motor.motor8.pack({"side":"left"})

 #
 # Boton para uso del objeto motor, uso del metodo calibraMotor
 #

 self.motores.motor.boton = Button(self.motores.motor,
                            {'text': 'ajustar a 0%',
                             'fg': 'blue',
                             'command' : self.ajustaMotor})
 self.motores.motor.boton.pack({"side":"left"})

 self.motores.motor.pack({"side" : "top"})
 # ACABA EL FRAME MOTORES.MOTOR

 #
 # Label que nos indica la funcion de los sig radiobotones
 #                       " Selecione el oclusion "
```

```
self.motores.label = Message(self.motores,
                        {"width" : "5i",
                         "text" : " Selecione el oclusion :"}
                        )

self.motores.label.pack(fill=X)


#
# Botoncitos de Oclusiones
#

self.motores.oclusion = Frame(self.motores)
# EMPIEZA EL FRAME MOTORES.OCLUSION
self.motores.oclusion.oclusion1 = Radiobutton (self.motores.oclusion,
                                {"text" : "mover 0 %",
                                 "variable" : self.oclusionx,
                                 "value" : 0,
                                 "anchor" : "w",
                                 Pack : {"side" : "top",
                                         "fill" : "x"}
                                 }
                                )
self.motores.oclusion.oclusion1.pack({"side":"left"})
self.motores.oclusion.oclusion2 = Radiobutton (self.motores.oclusion,
                                {"text" : "mover 25 %",
                                 "variable" : self.oclusionx,
                                 "value" : 1,
                                 "anchor" : "w",
                                 Pack : {"side" : "top",
                                         "fill" : "x"}
                                 }
                                )
self.motores.oclusion.oclusion2.pack({"side":"left"})
self.motores.oclusion.oclusion3 = Radiobutton (self.motores.oclusion,
                                {"text" : "mover 50 %",
                                 "variable" : self.oclusionx,
                                 "value" : 2,
                                 "anchor" : "w",
                                 Pack : {"side" : "top",
                                         "fill" : "x"}
                                 }
                                )
self.motores.oclusion.oclusion3.pack({"side":"left"})
self.motores.oclusion.oclusion4 = Radiobutton (self.motores.oclusion,
                                {"text" : "mover 75 %",
                                 "variable" : self.oclusionx,
                                 "value" : 3,
                                 "anchor" : "w",
                                 Pack : {"side" : "top",
                                         "fill" : "x"}
                                 }
                                )
self.motores.oclusion.oclusion4.pack({"side":"left"})
self.motores.oclusion.oclusion5 = Radiobutton (self.motores.oclusion,
                                {"text" : "mover 100 %",
                                 "variable" : self.oclusionx,
                                 "value" : 4,
                                 "anchor" : "w",
```

```
                                    Pack : {"side" : "top",
                                            "fill" : "x"}
                                    }
                                   )
self.motores.oclusion.oclusion5.pack({"side":"left"})

self.motores.oclusion.pack({"side" : "top"})
# ACABA EL FRAME MOTORES.OCLUSION

#
# Separador entre oclusion y calibra
#

self.motores.separador = Frame(self.motores)

self.motores.separador.label = Message(self.motores.separador,
                        {"width" : "5i",
                         "text" : ""}
                        )

self.motores.separador.label.pack(fill=X)

self.motores.separador.pack({"side" : "top"})

self.motores.pack({"side" : "top"})


#
# Label que nos indica la funcion de los sig radiobotones
#                       " Selecione la calibracion "

self.motores.label = Message(self.motores,
                        {"width" : "5i",
                         "text" : " Selecione la calibracion :"}
                        )

self.motores.label.pack(fill=X)



#
# Botoncitos de Calibracion
#

self.motores.calibrado = Frame(self.motores)
# EMPIEZA EL FRAME MOTORES.CALIBRADO
self.motores.calibrado.boton = Button(self.motores.calibrado,
                              {'text': 'Motor Calibrado',
                               'fg': 'blue',
                               'command' : self.calibraMotor})
self.motores.calibrado.boton.pack({"side":"left"})
self.motores.calibrado.pack({"side" : "left"})

self.motores.calibra = Frame(self.motores)
# EMPIEZA EL FRAME MOTORES.CALIBRA
self.motores.calibra.calibral = Radiobutton (self.motores.calibra,
                                {"text" : "Un paso atras",
                                 "variable" : self.oclusionx,
                                 "value" : 11,
                                 "anchor" : "w",
```

```
                                    Pack : {"side" : "top",
                                            "fill" : "x"}
                                    }
                                   )
self.motores.calibra.calibral.pack({"side":"left"})
self.motores.calibra.boton = Button(self.motores.calibra,
                                    {'text': 'Mover Motor',
                                     'fg': 'blue',
                                     'command' : self.mueveMotor})

self.motores.calibra.boton.pack({"side":"left"})
self.motores.calibra.calibra2 = Radiobutton (self.motores.calibra,
                                    {"text" : "Un paso adelante",
                                     "variable" : self.oclusionx,
                                     "value" : 10,
                                     "anchor" : "w",
                                     Pack : {"side" : "top",
                                             "fill" : "x"}
                                    }
                                   )
self.motores.calibra.calibra2.pack({"side":"left"})
self.motores.calibra.pack({"side" : "top"})
# EMPIEZA EL FRAME MOTORES.CALIBRA

#
# Separador entre motores y bomba
#

self.motores.separador = Frame(self.motores)

self.motores.separador.label = Message(self.motores.separador,
                        {"width" : "5i",
                         "text" : ""}
                        )

self.motores.separador.label.pack(fill=X)

self.motores.separador.pack({"side" : "top"})

self.motores.pack({"side" : "top"})
# ACABA EL FRAME MOTORES

#
# Dialogo de funccion de los radiobotones 5
#

self.bomba = Frame(self.top)
# EMPIEZA EL FRAME BOMBA
self.bomba.label = Message(self.top,
                        { "width" : "5i",
                        "text" : " Selecione el pulso : "}
                        )

self.bomba.label.pack(fill=X)

#
#Boton para configurar el pulso de la bomba "Activa Bomba"
#

self.bomba.izquierdo = Frame(self.bomba)
```

```
# EMPIEZA EL FRAME BOMBA.IZQUIERDO
self.bomba.izquierdo.boton = Button (self.bomba.izquierdo,
                            {"text" : "Activa Bomba",
                             'command' :
                             self.bombea,
                             Pack : {"side" : "top",
                                     "fill" : "x"}
                             }
                            )
self.bomba.izquierdo.boton.pack({"side":"left"})
self.bomba.izquierdo.pack({"side":"left"})
# ACABA EL FRAME BOMBA.IZQUIERDO

#
#Botoncitos de Pulso
#

self.bomba.derecho = Frame(self.bomba)
# EMPIEZA EL FRAME BOMBA.DERECHO
self.bomba.derecho.pulso1 = Radiobutton (self.bomba.derecho,
                                 {"text" : "50",
                                  "variable" : self.pulsox,
                                  "value" : 0,
                                  "anchor" : "w",
                                  Pack : {"side" : "top",
                                          "fill" : "x"}
                                  }
                                 )
self.bomba.derecho.pulso1.pack({"side":"right"})
self.bomba.derecho.pulso2 = Radiobutton (self.bomba.derecho,
                                 {"text" : "60",
                                  "variable" : self.pulsox,
                                  "value" : 1,
                                  "anchor" : "w",
                                  Pack : {"side" : "top",
                                          "fill" : "x"}
                                  }
                                 )
self.bomba.derecho.pulso2.pack({"side":"right"})
self.bomba.derecho.pulso3 = Radiobutton (self.bomba.derecho,
                                 {"text" : "70",
                                  "variable" : self.pulsox,
                                  "value" : 2,
                                  "anchor" : "w",
                                  Pack : {"side" : "top",
                                          "fill" : "x"}
                                  }
                                 )
self.bomba.derecho.pulso3.pack({"side":"right"})
self.bomba.derecho.pulso4 = Radiobutton (self.bomba.derecho,
                                 {"text" : "90",
                                  "variable" : self.pulsox,
                                  "value" : 3,
                                  "anchor" : "w",
                                  Pack : {"side" : "top",
                                          "fill" : "x"}
                                  }
                                 )
self.bomba.derecho.pulso4.pack({"side":"right"})
self.bomba.derecho.pulso5 = Radiobutton (self.bomba.derecho,
```

```
                                    {"text" : "120",
                                     "variable" : self.pulsox,
                                     "value" : 4,
                                     "anchor" : "w",
                                     Pack : {"side" : "top",
                                             "fill" : "x"}
                                     }
                                    )

self.bomba.derecho.pulso5.pack({"side":"right"})
self.bomba.derecho.pulso6 = Radiobutton (self.bomba.derecho,
                                    {"text" : "180",
                                     "variable" : self.pulsox,
                                     "value" : 5,
                                     "anchor" : "w",
                                     Pack : {"side" : "top",
                                             "fill" : "x"}
                                     }
                                    )

self.bomba.derecho.pulso6.pack({"side":"right"})

self.bomba.derecho.pack({"side":"left"})

# ACABA EL FRAME BOMBA.DERECHO
self.bomba.pack({"side" : "top"})
# ACABA EL FRAME BOMBA

#
# Separador entre bomba y monitor
#

self.monitor = Frame(self.top)
# EMPIEZA EL FRAME MONITOR
self.monitor.separador = Frame(self.monitor)
# EMPIEZA EL FRAME MONITOR.SEPARADOR
self.monitor.separador.label = Message(self.monitor.separador,
                        {"width" : "5i",
                         "text" : ""}
                        )

self.monitor.separador.label.pack(fill=X)

self.monitor.separador.pack({"side" : "top"})
# EMPIEZA EL FRAME MONITOR.SEPARADOR

#
# Dialogo de funccion de los Botones monitor
#


self.monitor.label = Message(self.monitor,
                       { "width" : "5i",
                       "text" : " Selecione el monitor :"})
self.monitor.label.pack(fill=X)

#
#Botonctitos de Monitor
#
```

```
self.monitor.AID = Frame(self.monitor)
# EMPIEZA EL FRAME MONITOR.AID
self.monitor.AID.ok = Button (self.monitor.AID,
                        {"text" : "Activa Monitor",
                         'command' :
                         self.grafica,
                         Pack : {"side" : "top",
                                 "fill" : "x"}
                         }
                        )
self.monitor.AID.ok.pack({"side":"left"})
self.monitor.AID.monitor1 = Radiobutton (self.monitor.AID,
                            {"text" : "monitor1",
                             "variable" : self.monitorx,
                             "value" : "0",
                             "anchor" : "w",
                             Pack : {"side" : "top",
                                     "fill" : "x"}
                             }
                            )

self.monitor.AID.monitor1.pack({"side":"left"})
self.monitor.AID.monitor2 = Radiobutton (self.monitor.AID,
                            {"text" : "monitor2",
                             "variable" : self.monitorx,
                             "value" : "1",
                             "anchor" : "w",
                             Pack : {"side" : "top",
                                     "fill" : "x"}
                             }
                            )
self.monitor.AID.monitor2.pack({"side":"left"})
self.monitor.AID.monitor3 = Radiobutton (self.monitor.AID,
                            {"text" : "monitor3",
                             "variable" : self.monitorx,
                             "value" : "2",
                             "anchor" : "w",
                             Pack : {"side" : "top",
                                     "fill" : "x"}
                             }
                            )
self.monitor.AID.monitor3.pack({"side":"left"})
self.monitor.AID.monitor4 = Radiobutton (self.monitor.AID,
                            {"text" : "monitor4",
                             "variable" : self.monitorx,
                             "value" : "3",
                             "anchor" : "w",
                             Pack : {"side" : "top",
                                     "fill" : "x"}
                             }
                            )
self.monitor.AID.monitor4.pack({"side":"left"})
self.monitor.AID.monitor5 = Radiobutton (self.monitor.AID,
                            {"text" : "monitor5",
                             "variable" : self.monitorx,
                             "value" : "4",
                             "anchor" : "w",
                             Pack : {"side" : "top",
                                     "fill" : "x"}
                             }
```

```
                                )
self.monitor.AID.monitor5.pack({"side":"left"})
self.monitor.AID.monitor6 = Radiobutton (self.monitor.AID,
                               {"text" : "monitor6",
                                "variable" : self.monitorx,
                                "value" : "5",
                                "anchor" : "w",
                                Pack : {"side" : "top",
                                        "fill" : "x"}
                                }
                               )
self.monitor.AID.monitor6.pack({"side":"left"})
self.monitor.AID.monitor7 = Radiobutton (self.monitor.AID,
                               {"text" : "monitor7",
                                "variable" : self.monitorx,
                                "value" : "6",
                                "anchor" : "w",
                                Pack : {"side" : "top",
                                        "fill" : "x"}
                                }
                               )
self.monitor.AID.monitor7.pack({"side":"left"})
self.monitor.AID.monitor8 = Radiobutton (self.monitor.AID,
                               {"text" : "monitor8",
                                "variable" : self.monitorx,
                                "value" : "7",
                                "anchor" : "w",
                                Pack : {"side" : "top",
                                        "fill" : "x"}
                                }
                               )
self.monitor.AID.monitor8.pack({"side":"left"})

self.monitor.AID.pack({"side":"top"})
# ACABA EL FRAME MONITOR.AID

#
# Dialogo de funccion de los Botones monitor
#

      self.monitor.regla = Frame(self.monitor)
# EMPIEZA EL FRAME MONITOR.REGLA
self.monitor.regla.label = Message(self.monitor.regla,
                     { "width" : "5i",
                     "text" :
                     " Selecione el tiempo de monitoreo :"})

self.monitor.regla.label.pack()

#
# Ajuste de tiempo
#

self.monitor.regla.slider = Scale(self.monitor.regla, {"from" : 5,
                              'to': 420,
                              "orient" : "horizontal",
                              "length" : "8i",
                              "label" :
                              "Tiempo de monitoreo",
                              'command' :
```

```
                                        self.tiempoGuarda,
                                        Pack : {"side" : "bottom",
                                                "fill" : "x"}
                                        )
                                )

        self.monitor.regla.slider.pack({"side":"top"})

        self.monitor.regla.pack({"side":"bottom"})
        # ACABA EL FRAME MONITOR.REGLA
        self.monitor.pack({"side":"top"})
        # ACABA EL FRAME MONITOR

        #
        # Bye Chao Adios Aurevoir
        #

        self.bye = Frame(self.top)
        # EMPIEZA EL FRAME BYE
        self.bye.boton = Button(self.top, {'text': 'QUIT',
                                           'fg': 'blue',
                                           'command' : self.quit})
        self.bye.boton.pack(side=BOTTOM, expand=YES, fill=BOTH)
        self.bye.pack({"side":"top"})
        # ACABA EL FRAME BYE

    # Metodo tiempoGuarda, almacena en una variable el valor selecionado en el
    # " slider "
    # Instancia.tiempoGuarda(Integer)

    def tiempoGuarda(self,val) :
      if self.verDebug() == 1 :
          print val
      self.tiempox.set(val)

    # Metodo mueveMotor, nos idica que motor deseamos mover y invoca el metodo
    # activaMotor del objeto motor en cuestion con la posicion deseada
    # Instancia.mueveMotor()
    def mueveMotor(self) :
      if self.verDebug() == 1 :
          print " La instruccion que se ejecutara es activaMotor"
          print " activaMotor con id ",self.motorx.get()," en posicion ",
self.oclusionx.get()
      self.M.activaMotor(self.motorx.get(), self.oclusionx.get())

    # Metodo ajustaMotor, nos idica que motor deseamos ajustar a 0% y invoca el
    # metodo ajustaMotor del objeto motor en cuestion
    # Instancia.ajustaMotor()

    def ajustaMotor(self) :
      if self.verDebug() == 1 :
          print " La instruccion que se ejecutara es ajusta el motor"
          print " ajustaMotor con id ",self.motorx.get()
      self.M.ajustaMotor(self.motorx.get())

    # Metodo grafica, nos indica que canal y cuanto tiempo lo vamos a monitorear
    # y invoca el metodo grafica del monitor instanciado
    # Instancia.grafica()

    def calibraMotor(self) :
```

```
      if self.verDebug() == 1 :
          print " La instruccion que se ejecutara es calibra el motor"
          print " CalibraMotor con id ",self.motorx.get()
      self.M.calibraMotor(self.motorx.get())

    # Metodo grafica, nos indica que canal y cuanto tiempo lo vamos a monitorear
    # y invoca el metodo grafica del monitor instanciado
    # Instancia.grafica()

    def grafica(self) :
      if self.verDebug() == 1 :
          print " La instruccion que se va a ejecutar es grafica"
          print " Se activara el canal ",self.monitorx.get()," por ",
self.tiempox.get()," seg"
      self.G.activaMonitor(self.monitorx.get(),self.tiempox.get())

    # Metodo bombea, nos indica el ritmo al cual va a pulsar la bomba
    # y invoca el objeto bomba con la opcion correspondiente

    def bombea(self) :
      if self.verDebug() == 1 :
          print " La instruccion que se va a ejecutar es bombea"
          print " La opcion del ritmo es ",self.pulsox.get()
      self.B.activaBomba(self.pulsox.get())

      # Se ejecuta cuando se instancia la clase Monitor
      # Se instancia los Motores, la Bomba, el Monitor y se  crea los Widgets
      # de la interface grafica


    def __init__(self, master=None):
      # Se detecta si el usuario invoca el programa en modo normal
      # o de depuracion
      try:
          if sys.argv[1] == "-d" :
            self.cambiarDebug()
            print "El programa se encuentra en modo de depuracion"
      except IndexError :
          print "El programa se encuentra en modo normal"
      # Se instancia los "Motores" que vamos a utilizar
      self.M=actuadores.Motores()
      # Se instancia la "Bomba" que vamos a utilizar
      self.B=actuadores.Bomba()
      # Se instancia el "Monitor" que vamos a utilizar
      self.G=monitor.Monitor()
      Frame.__init__(self, master)
      Pack.config(self)
      self.pack()
      self.creaWidgets()

    # Metodo que le inica a python como instancia la interface grafica
```

```
def inicia():
    app=Gui().mainloop()

    # Metodo que hace invocar inicia() si se ejecuta :
    # python " El nombre de este modulo "

if __name__ == '__main__':
    inicia()
```

```
# Nombre de modulo: bandera
# Clases dentro del modulo: 1
#                               - Bandera

#Clase que se hereda a scc, actuadores, monitor, serial

class Bandera :

     # Atributo de clase para indicarle al usuario cuando es necesario que
     #  reinicialize el microcontrolador

     status = 0
     def cambiarStatus(self,var):
       Bandera.status = var
     def verStatus(self):
       return Bandera.status

     # Atriuto de clase para indicar a los objetos que la herdan
     # si se desea que se desplieguen los mensajes de depuracion

     debug = 0
     def cambiarDebug(self):
       Bandera.debug = 1
     def verDebug(self):
       return Bandera.debug
```

```
# Nombre de modulo: monitor
# Clases dentro del modulo: 1
#                           - Monitor

# Importacion de modulos necesarios


import posix            # Sirve para configurar el puerto serial
import os               # Sirve para manejar el file descriptor del puerto serial
import FCNTL            # Sirve para configurar el file descriptor del puerto serial
import time             # Sirve para medir el tiempo que se despliega el monitor
import comunicacion     # Sirve para mandarle instrucciones al microcontrolador
from string import *    # Sirve para manejar los datos recibidos por puerto serial
from Numeric import *   # Sirve para crear la matriz de datos necesaria para graficar
from gist import *      # Sirve para graficar los datos del puerto serial

# Clase Monitor, sirve para graficar en un plano cartesiano los datos que se
# reciben del microcontrolador por el puerto serial.

class Monitor:

    # Metodo grafica, despliega por un cierto tiempo los datos que le manda el
    # microcontrolador
    # Instacia.grafica(Integer)

    def grafica(self,duracion):

        ############### CREACION DEL OBJETO RSER PARA LEER EL PUERTO SERIAL ########
      # Configuracion de puerto serial
      posix.system('stty
1401:0:cbd:0:3:1c:7f:15:4:5:1:0:11:13:1a:0:12:f:17:16:0:0:73
< /dev/ttyS1')
      # Apertura de puerto serial
      rserial = os.open("/dev/ttyS1" ,
                        FCNTL.O_RDONLY)
      # Creacion de file descriptor de puerto serial
      rser = os.fdopen(rserial, 'r')

      ############### GRAFICACION ###################
      # Creacion de ventana donde se va a graficar
      window(style="work.gs")
      # Indicacion de limites de graficacion
      limits(0,56,0,255)
      # Indicacion de titulo de la ventana de graficacion
      pltitle("Monitor")
      # raw_input nesesario para coordinar el Tk y el gist
      raw_input()
      # Paso a modo de graficacion animate donde se despliega la grafica hasta
      # que esta lista
      animate()
      # Iniciacion de i : Mariable de el loop de for donde se almacenan los datos
      #                   del puerto serial en un matriz de forma especifica
      #                l0: Matriz de los datos del puerto serial; representa un
      #                   cuadro
      i = 0
      l0 = []
      # Se guarda en valor el tiempo en que se debe de para la ejecucion
      stop = time.time() + duracion
      # Creacion de la matriz de datos a desplegar
      for i in range(112) :
```

```
        10.append(0)
        a = array(range(56),"b")
        while time.time() < stop :
          b = array([])
          i = rser.read(112)
          s = [atoi(i[0:2],16),     atoi(i[2:4],16),      atoi(i[4:6],16),
               atoi(i[6:8],16),     atoi(i[8:10],16),     atoi(i[10:12],16),
               atoi(i[12:14],16),   atoi(i[14:16],16),    atoi(i[16:18],16),
               atoi(i[18:20],16),   atoi(i[20:22],16),    atoi(i[22:24],16),
               atoi(i[24:26],16),   atoi(i[26:28],16),    atoi(i[28:30],16),
               atoi(i[30:32],16),   atoi(i[32:34],16),    atoi(i[34:36],16),
               atoi(i[36:38],16),   atoi(i[38:40],16),    atoi(i[40:42],16),
               atoi(i[42:44],16),   atoi(i[44:46],16),    atoi(i[46:48],16),
               atoi(i[48:50],16),   atoi(i[50:52],16),    atoi(i[52:54],16),
               atoi(i[54:56],16),   atoi(i[56:58],16),    atoi(i[58:60],16),
               atoi(i[60:62],16),   atoi(i[62:64],16),    atoi(i[64:66],16),
               atoi(i[66:68],16),   atoi(i[68:70],16),    atoi(i[70:72],16),
               atoi(i[72:74],16),   atoi(i[74:76],16),    atoi(i[76:78],16),
               atoi(i[78:80],16),   atoi(i[80:82],16),    atoi(i[82:84],16),
               atoi(i[84:86],16),   atoi(i[86:88],16),    atoi(i[88:90],16),
               atoi(i[90:92],16),   atoi(i[92:94],16),    atoi(i[94:96],16),
               atoi(i[96:98],16),   atoi(i[98:100],16),   atoi(i[100:102],16),
               atoi(i[102:104],16),atoi(i[104:106],16), atoi(i[106:108],16),
               atoi(i[108:110],16),atoi(i[110:112],16)]
          b = array(s,"b")
          # Instruccion de frame advance, evita el parpadeo en el despliegue
          fma()
          # Graficacion de matriz de datos
          pldj(a[:-1],b[:-1],a[1:],b[1:])
    # Desactivacion del modo de animacion despues de el tiempo
    animate(0)
    # Graficacion de la utlima matriz de datos
    pldj(a[:-1],b[:-1],a[1:],b[1:])
    # Se espera  que el usuario pulse cualquier tecal cuando quiera salir
    raw_input()
    # Destruccion de cuadro de graficacion
    winkill(0)

# Metodo activaMonitor, configura el microcontrolador para que mande los datos
# que cuantiza en uno de sus canales del convertidor AID y lanza la subrutina
# de muestreo y transmision de datos
# Instancia.activaMonitor(String,Integer)

def activaMonitor(self, monitorx, duracion):

      #### EJECUCION DE METODO Y SUBRUTINA DEL HC11 PARA DESPLEGAR EL MONITOR ###
    # Iniciacion del puerto serial
    print " Presione 1 y enter para graficar "
    self.C.iniciacion()
    time.sleep(0.1)
    # Configuracion de que canal del A/D del microcontrolador se va a muestrar
    self.C.envia("mm 108")
    time.sleep(0.1)
    self.C.envia(monitorx)
    time.sleep(0.1)
    # Ejecucion de subrutina de muestreo del microcontrolador
    self.C.envia("go b65b")
    time.sleep(0.1)
    # Ejecucion de metodo que despliega los datos recolectados en el puerto
    # serial
```

```
    print " Presione cualquier tecla para continuar"
    self.grafica(duracion)


# Se ejecuta cuando se instancia la clase Monitor

def __init__(self):
    print "Estoy inicializando la clase serial en instancia self.C "
    # Se instancia la "comunicacion" que vamos a utilizar
    self.C=comunicacion.Serial()
    # Se pone a 0 la variable de clase status para senalar que es necesaria
    # una reinciacion
    self.C.cambiaPonerStatus(0)
```

```
# Nombre de modulo : comunicacion
# Clases dentro del modulo : 1
#                                - Serial

# Importacion de funciones necesarias


import termios    # Sirve para manejar el file descriptor del puerto serial
import TERMIOS    # Sirve para manejar el file descriptor del puerto serial
import posix      # Sirve para configurar el puerto serial
import os         # Sirve para manejar el file descriptor del puerto serial
import FCNTL      # Sirve para configurar el file descriptor del puerto serial
import bandera     # Sirve para heredar el atributo de clase de control de debug

# Clase Serial, sirve para accesar el puerto serial, enviar instruciones al
# microcontorlador a traves del puerto serial y recibir mensajes del
# microcontrolador por medio del metodo envia

class Serial(bandera.Bandera):


    # Metodo envia, manda a traves del puerto serial el string deseado al
    # microcontrolador
    # Instancia.envia("sting")

    def envia(self,instruccion):

      # Se le anade salto de linea a la instruccion para validarla
      self.abreFdLectura()
      self.instruccion = instruccion + "\r"
      # Se envia la instruccion al microcontrolador
      if self.verDebug() == 1 :
          print "La bandera esta en ",self.verStatus()
      self.fdEscritura.write(self.instruccion)
      # Se recibe el mensaje que nos delvuelve el microcontrolador
      mensaje1 = self.fdLectura.readline(200)
      mensaje2 = self.fdLectura.readline(200)
      mensaje3 = " "
      mensaje4 = " "
      if instruccion == "go b600" :
          if self.verDebug() == 1 :
            print "Leyendo un tercer y cuarto mensaje"
          mensaje3 = self.fdLectura.readline(200)
          mensaje4 = self.fdLectura.readline(200)
      if instruccion == "go b65b" :
          mensaje1 = ""
          mensaje2 = ""
      # Se despliega el mensaje que nos mando el microcontrolador
      if self.verDebug() == 1 :
          print(mensaje1+mensaje2+mensaje3+mensaje4)
      # Se cierran los file descriptos de los atributos de lectura y
      # escritura
      self.fdLectura.close()
      # Cambia al valor asignado a ponerStatus, el atributo de clase status
      self.cambiarStatus(self.ponerStatus)
      if self.verDebug() == 1 :
          print " Se puso la bandera a ",self.ponerStatus
```

```
# Metodo inciacion, vacia el buffer de la computadora, pone en linea
# el microcontrolador,  y inicializa una direcciones de memoria del micro.

def iniciacion(self):

  # Se checa el valor de la bandera status para ver si se procede a una
  # inicializacion
  if self.verStatus() == 0 :
        #Inicializacion con la primer instruccion mandada
      #raw_input('Inicialize su microprocesador y pulse aceptar')
      #print " Buffer de la computadora vacio "
      raw_input('Inicialize su microprocesador y pulse enter')
      self.abreFdLectura()
      termios.tcflush(self.fdLectura.fileno(),TERMIOS.TCIFLUSH)
      print "inicialize el micro"
      mensaje = self.fdLectura.read(70)
      print(mensaje)
      # String de iniciacion del microcontrolador que necesita la
      # primera ves que se comunica con el
      self.envia('\r  bf 0100 0108 00')
  else :
      pass


# Metodo CambiaPonerStatus, methodo para indicarle al objeto instanciado
# si debe de poner a uno o a cero el atributo de clase, status.

def cambiaPonerStatus(self,var):
  self.ponerStatus = var

  # Metodo, abreFdLectura, apertura de objeto de lectura rser
  # abreFdLectura()

def abreFdLectura(self):
  # creacion del fd para leer en el microcontrolador
  rserial = os.open("/dev/ttyS1" ,
                    FCNTL.O_RDONLY)
  # Instanciacion de el fd para lerr en el fd
  self.fdLectura = os.fdopen(rserial, 'r')
  # Se detiene el programa para estar seguro que esta resetiado el
  # microcontrolador

# Se ejecuta cuando se instancia la clase Monitor
```

```
    def __init__(self):

      # Instanciacion de objeto para que siempre asigne 1 a la bandera status
      self.ponerStatus = 1
      # Configuracion del puerto serie mediante el comando tty y los
      # parametros sacados de minicom
      posix.system('stty
1401:0:cbd:0:3:1c:7f:15:4:5:1:0:11:13:1a:0:12:f:17:16:0:0:73 < /dev/ttyS1')

      # Apertura de objeto de escritura wser p

      # Creacion del file descriptor para escribir en el microcontrolador
      wserial = os.open("/dev/ttyS1" ,
                        FCNTL.O_WRONLY|FCNTL.O_NONBLOCK|FCNTL.O_SYNC)
      # Instanciacion de el file descriptor (fd) de escritura
      self.fdEscritura = os.fdopen(wserial, 'w')
      self.iniciacion()
```

```
# Nombre de modulo: actuadores
# Clases dentro del modulo: 3
#                                - Bomba
#                                - Motores
#                                - Motor

# Importacion de modulos necesarios

import comunicacion    # Sirve para mandarle instrucciones al microcontrolador
import time            # Sirve para separar cada instrucion que se manda al micro
import jiuuuu          # Sirve para heredar el atributo de clase de control de debug
import bandera         # Sirve para heredar el atributo de clase de control de debug

# Clase Bomba, sirve para activar la bomba al pulso deseado
#

class Bomba:

     # Metodo activaBomba, configura el ritmo de la Bomba
     # Instancia.activaBomba(Integer)

     def activaBomba(self, pulso):

       # Declaracion del diccionario bomba nos regresa el sting que se debe de
       # enviar al controlador segun la llave de tipo entero.
       codigosPulso={0:'0C',1:'0A',2:'08',3:'06',4:'04',5:'02'}
       pulsoPorMinuto={0:'50',1:'60',2:'70',3:'80',4:'120',5:'180'}
       # Se almacena en pulso el string que se debe de enviar al
       # microcontrolador segun la opcion de pulso que se seleciono en el GUI
       pulsoProgramado = codigosPulso[pulso]
       print "La Bomba se configura a",pulsoPorMinuto[pulso],"pulsaciones por
minuto"
       # Envio de instrucciones al microcontrolador
       self.C.iniciacion()
       time.sleep(0.2)
       self.C.envia("mm 107")
       time.sleep(0.2)
       self.C.envia(pulsoProgramado)
       time.sleep(0.2)
       self.C.envia("mm 1004")
       time.sleep(0.2)
       self.C.envia(pulsoProgramado)

     # Se ejecuta cuando se instancia la clase Bomba

     def __init__(self):
       print "Estoy inicializando la clase serial en instancia self.C "
       # Se instancia la "comunicacion" que vamos a utilizar
       self.C=comunicacion.Serial()

Id=['1c','18','14','10','0c','08','04','00']
Sentido={'adelante':'10','atras':'20'}


# Clase Motor, contiene la estructura de datos abstracta de un motor, con un metodo
# para cambiar la olclusion de este

class Motor(bandera.Bandera):

     # Metodo cambiaOclusion, sirve para cambiar la oclusion en la cual se encuentra
```

```
# un objeto motor ademas contiene un atributo que indica el sentido y el
# numero de pasos que tiene que dar el motor para llegar a la posicion deseada
# Instancia.cambiaOclusion(Integer)

def cambiaOclusion(self,posicion) :
  if posicion < 10 :
       dif = posicion - self.oclusion
       if dif > 0 :
         self.sentido=Sentido['adelante']
         if self.verDebug() == 1 :
              print " El sentido que se va tomar es hacia adelante "
       elif dif < 0 :
         self.sentido=Sentido['atras']
         if self.verDebug() == 1 :
              print " El sentido que se va tomar es hacia atras "
       else:
         self.sentido=Sentido['adelante']
       paso = int(self.carrera/5*dif)
  else:
       if posicion == 10 :
         paso = 1
         self.sentido=Sentido['adelante']
         if self.verDebug() == 1 :
              print " El motor da un paso hacia adelante "
         self.carreraCalibrada = self.carreraCalibrada + 1
       elif posicion == 11 :
         paso = 1
         self.sentido=Sentido['atras']
         if self.verDebug() == 1 :
              print " El motor da un hacia atras "
         self.carreraCalibrada = self.carreraCalibrada - 1
  if self.verDebug() == 1 :
       print " El numero de pasos que se va a dar ",paso
  self.pasos = hex(abs(paso))
  if self.verDebug() == 1 :
       print " El numero de pasos que se daran en hexadecimal ",self.pasos[2:]
       print " La Carrera que se tiene es de ",self.carrera
       print " La Carrera calibrada que se tiene es ", self.carreraCalibrada
  self.oclusion = posicion

# Metodo calibra, asigna al atributo  carreraCalibrada el valor del atributo
# carrera que se incrementa paso a paso cuando el usuario calibra el motor

def calibra(self) :
  print "Motor calibrado a ",self.carreraCalibrada
  if self.verDebug() == 1 :
       print " La Carrera que se tenia era de ",self.carrera
       print " La Carrera calibrada que desea es", self.carreraCalibrada
  self.carrera = self.carreraCalibrada
  self.oclusion = 4
  if self.verDebug() == 1 :
       print " La Carrera Nueva que se tiene es de ",self.carrera


# Se ejecuta cuando se instancia la clase Bomba

def __init__(self,sentido,oclusion,identidad):
  self.sentido = sentido
  self.oclusion = oclusion
  self.identidad = identidad
```

```
        self.carrera = 20
        self.carreraCalibrada = 0

# Clase Motores, Contiene en una lista las instancia de los motores, y los metodos
# para activar y calibrar los motores

class Motores(bandera.Bandera):

    # Metodo calibraMotor, guarda en la intancia de motor seleccionada, el numero
    # de pasos necesario para que el embolo ocluya 100% la arteria.
    # Instancia.calibraMotor(Integer)

    def calibraMotor(self,opcion_motor):
      self.listaMotores[opcion_motor].calibra()

    # Metodo activaMotor, manda al microcontorlador las instrucciones adecuadas
    # para que se desplaze el motor a la posicion deseada.
    # Instancia.activaMonitor(Integer, Integer)

    def activaMotor(self, opcion_motor, opcion_oclusion) :

      if self.verDebug() == 1 :
          print " Se checa que objetito instanciado"
          print " El motor ",self.listaMotores[opcion_motor].identidad
      oclusionAnterior = self.listaMotores[opcion_motor].oclusion
      if self.verDebug() == 1 :
          print " Tenia oclusion de",self.listaMotores[opcion_motor].oclusion
          print " Su sentido es ",self.listaMotores[opcion_motor].sentido
          print " Se configura el motorsito"
      self.listaMotores[opcion_motor].cambiaOclusion(opcion_oclusion)
      if self.verDebug() == 1 :
          print " Se checa que objetito nos sale de nuez"
          print " Su identidad es ",self.listaMotores[opcion_motor].identidad
          print " Se movera a oclusion",self.listaMotores[opcion_motor].oclusion
          print " Con un sentido de ",self.listaMotores[opcion_motor].sentido
      if oclusionAnterior != self.listaMotores[opcion_motor].oclusion
or oclusionAnterior > 9 :
          print "Moviendo motor",opcion_motor
          self.C.iniciacion()
          time.sleep(0.2)
          self.C.envia("mm 100")
          time.sleep(0.2)
          print "*"
          #***** Se escoje el motor *****
          self.C.envia(self.listaMotores[opcion_motor].identidad)
          time.sleep(0.2)
          self.C.envia("mm 101")
          time.sleep(0.2)
          print "*"
          #***** Se escoje el sentido *****
          self.C.envia(self.listaMotores[opcion_motor].sentido)
          time.sleep(0.2)
          self.C.envia("mm 102")
          time.sleep(0.2)
          self.C.envia("00")
          time.sleep(0.2)
          self.C.envia("mm 103")
          time.sleep(0.2)
          print "*"
          #***** Se escoje el numero de pasos *****
```

```
        self.C.envia(str(self.listaMotores[opcion_motor].pasos[2:]))
        time.sleep(0.2)
        self.C.envia("mm 104")
        time.sleep(0.2)
        self.C.envia("5F")
        time.sleep(0.2)
        self.C.envia("mm 105")
        time.sleep(0.2)
        self.C.envia("FF")
        time.sleep(0.2)
        self.C.envia("go b600")
    else :
        print " No se envia nada al microcontraldor dado que la oclusion en el"
        print " motor selecionado no cambia "

# Metodo ajustaMotor, retrocede el motor selecionado a la poscicion cero
# Instancia.ajustaMotor(Integer)

def ajustaMotor(self,opcion_motor) :
    print " Ajustando motor a 0"
    if self.verDebug() == 1 :
        print " Se checa que objetito instanciado"
        print " Su identidad es ",self.listaMotores[opcion_motor].identidad
        print " Su oclusion es ",self.listaMotores[opcion_motor].oclusion
        print " Su sentido es ",self.listaMotores[opcion_motor].sentido
        print " Se configura el motorsito"
        print "* Se pone a cero la estructura de datos del motor en cuestion *"
    self.listaMotores[opcion_motor].cambiaOclusion(00)
    if self.verDebug() == 1 :
        print " Se checa que objetito nos sale de nuez"
        print " Su identidad es ",self.listaMotores[opcion_motor].identidad
        print " Su oclusion es ",self.listaMotores[opcion_motor].oclusion
        print " Su sentido es ",self.listaMotores[opcion_motor].sentido
        print " Envio de instrucciones "
    self.C.iniciacion()
    time.sleep(0.2)
    if self.verDebug() == 1 :
        print "********  Se arma el paro automatico por hardware **********"
    self.C.envia("mm 106")
    time.sleep(0.2)
    self.C.envia("10")
    self.C.envia("mm 100")
    time.sleep(0.2)
    print "*"
    #***** Se escoje el motor *****
    self.C.envia(self.listaMotores[opcion_motor].identidad)
    time.sleep(0.2)
    self.C.envia("mm 101")
    time.sleep(0.2)
    self.C.envia("20")
    time.sleep(0.2)
    self.C.envia("mm 102")
    time.sleep(0.2)
    self.C.envia("00")
    time.sleep(0.2)
    self.C.envia("mm 103")
    time.sleep(0.2)
    self.C.envia("20")
    time.sleep(0.2)
    self.C.envia("mm 104")
```

```
    time.sleep(0.2)
    self.C.envia("5F")
    time.sleep(0.2)
    self.C.envia("mm 105")
    time.sleep(0.2)
    self.C.envia("FF")
    time.sleep(0.2)
    self.C.envia("go b600")

    # Se ejecuta cuando se instancia la clase Monitor
      # Creacion de la lista de los motores

def __init__(self):
    if self.verDebug() == 1 :
        print "Estoy inicializando la clase serial en instancia self.C "
    # Se instancia la "comunicacion" que vamos a utilizar
    self.C=comunicacion.Serial()
    # Se define la lista Motores de objetos instanciados indexados motor
    self.listaMotores=[]
    for i in Id:
        motor=Motor(Sentido['adelante'],0,i)
        self.listaMotores.append(motor)
```

Glosario

A

Aneurisma	Tumor sanguíneo formado por la dilatación de las paredes de una arteria.
Angiografía	Descripción de los vasos sanguíneos y linfáticos.
ANSI/IS	American National Standards Institute / International Standards Organization.
API	Interface para programación de aplicaciones. Puede ser la interface a una biblioteca o a un sistema.
Arteriosclerosis	Engrosamiento y endurecimiento de las arterias, cuyas paredes muestran degeneración celular y en ocasiones calcificación.

B

Bradicardia	Ritmo excesivamente lento de la contracción cardiaca.

C

CGM	Computer Graphics Metafile.

D

Drive	Unidad de lectura de discos flexibles en una computadora.

F

$FICO_2$	Fracción de bióxido de carbono inhalada en la respiración.
Fibrilación auricular	Vibración arrítmica de las fibras musculares de las aurículas del corazón.
FIO_2	Fracción de oxígeno inhalada en la respiración.
Fluido pseudoplástico	Fluido que mantiene en forma semiconstante su forma.
Fluido viscoelástico	Fluido que mantiene en forma constante sus propiedades de tensión, resistencia y dirección de movimiento.

G

Gist	Biblioteca de graficación científica.
GUI	Interface gráfica de usuario.

I

Infarto	Suspensión del riego sanguíneo en alguna región, causada por la oclusión de una arteria.

K

Kernel	Interface de entrada/salida entre el hardware y el manejador de memoria.

L

Linux	Sistema operativo multiplataforma compatible con sistemas UNIX.

M

Mesh	Tipo de Gráfica.
Método no invasivo	Hablando de medición de signos vitales, es todo aquel método que realiza dichas mediciones en forma externa al ser humano, es decir, sin introducir ningún agente extraño al cuerpo.
Miocardio	Parte musculosa del corazón.
Motif	Bibliotecas de graficación.

N

Numeric	Módulo de Python para manejo de operaciones numéricas o computacionalmente intensivas.

P

$PACO_2$	Presión parcial del bióxido de carbono en la sangre de las arterias.
Python	Lenguaje de programación interpretado con orientación a objetos.
Polígono de Willis	Arteria localizada en la base del cerebro. Está encargada del suministro sanguíneo a arterias secundarias y vasos en el cerebro.
Postcript	Formato gráfico.
Purgar	Eliminar el aire.

R

RedHat 2.02	Distribución de un Sistema Operativo con un kenel de Linux.

T

TCL/TK	Lenguaje de programación con widgets para crear Interfaces Gráficas de Usuario.
TKinter	Módulo de acoplamiento entre TK/TCL y Python.
Toolkit	Conjunto de herramientas para desarrollo de software.
Trombo	Coágulo de sangre que se forma en un vaso sanguíneo o en el interior de una de las cavidades del corazón.

W

Widget	Clases compuestas de elementos predefinidos como marcos, botones, etc.

X

Xlib	Biblioteca de graficación.
X-Windows	Servidor para despliegue gráfico sobre plataformas Unix.

Bibliografía y referencias

- Ascher, David. *Python a research toolkit for scientists and engineers.* Diapositivas de una presentación en Python Workshop 5 disponible en *www.python.org/workshops/1996-11/.*
- Boyce, Jefferson. *Operational Amplifiers and Linear Integrated Circuits.* PWS-KENT Publishing Company.
- Coad, Peter and Jill, Nicolas. *Object-Oriented Programing.* Yourdon Press.
- Conway, Matt. *A Tkinter Life Preserver.* Manual de referencia para la interface entre Python y el GUI de Tk
- Felsinger, Richard C. *Object-Oriented Modeling using With Class* Tutorial de paquete de CASE With Class disponible en *www.microgold.com.*
- Geddes, L.A. y L.E. Baker. *Principles of Applied Biomedical Instrumentation.* Wiley-Interscience Publication.
- Ghezzi, Carlo and Jazayeri Mehdi and Mandrioli Dino. *Software Engineering.* Pentice Hall
- Ibargüengoitia Gonzáles, Ma. Guadalupe. *INGENIERIA DE SOFTWARE Notas de clase.* Maestría en Ciencias de la Computación UACPyP del CCH / IIMAS -UNAM.
- Jones, Douglas W. *Control of Stepping Motors, a tutorial.* Disponible en *http ://www.cs.uiowa.edu/~jones/step*
- Lutz, Mark. *Programming Python.* O'Reilly & Associates, Inc.
- Manheimer (original), Ken, Hoffmann Chris y Baxter Anthony. *Quick Reference Documents.* Disponibles en *ftp.python.org/pub/python/doc.*
- Motorola Inc. *HC11. MC68HC711E9 Technical Data.*
- Motorola Inc. *MC68HC11E9 (Advance Info.) HCMOS Single-Chip Microcontroller.*
- Motorola Inc. *M68HC11EVBU Universal Evaluation Board User's Manual.*
- Motorola Inc. *Sensor Device Data.*
- Paul F.Dubois, Konrad Hisen and James Hugunin. *Numerical Python.* Disponible en *ftp-icf.llnl.gov/pub/basis/numerical_python.ps*
- Protel Technology. *Advanced Schematic 2 User Guide / Reference* . Disponible en *www.protel.com/download.html*
- Protel Technology. *Advanced PCB On-line Reference* . Disponible en *www.protel.com/download.html*
- Segal, Bernard. *Engineering in the Practice of Medicine.* The Williams & Wilkins Co.

• Van Rossum, Guido. *Python Reference Manuals.*
Un tutorial, referencia de lenguage, referencia de bibliotecas y referencia para extender python, están disponibles por FTP en *ftp.python.org* o en la WWW en *www.python.org*.

[1] Clynes, Manfred. *Biomedical Engineering Systems.* McGraw-Hill

[2] Brian W. Chong . *Blood Flow Dynamics in the Vertebrobasilar System : Correlation of a Transparent Elastic Model and MR Angiography.* AJNR 15 :733-745, Apr. 1994

[3] Dieter Liepsch . *Fundamental Flow Studies in Models of Human Arteries.*_Frontler Med. Biol. Engng., Vol. 5, No. 1, pp. 51 - 55 (1993)

[4] Sandro Rossitti . *Vascular Dimensions of the cerebral Arteries Follow the Principle of Minimum Work.*_Stroke. Vol 24, No. 3 March 1993

[5] C.E. Elwell . *An automated System for the Measurement of the Response of Cerebral Blood Flow to Changes in Arterial Carbon Dioxide Tension Using Near Infared Spectroscopy.*_Biol. Eng & Comp., 26, 3 : 289 - 294 (1988)

[6] National Instruments Co. *National Instruments Data Acquisition Seminar.*

[7] Geddes, L.A., L.E. Baker. *Principles of Applied Biomedical Instrumentation.* Wiley-Interscience Publication. 3er. Ed

[8] Javier Cárdenas Flores . *Python : Un lenguage Orientado a Objetos alternativo.* Soluciones Avanzadas Año 5 Número 39 (15 Nov 1996)

[9] Coad, Peter and Jill, Nicolas . *Object-Oriented Programing.* Yourdon Press.