Universidad Carlos III de Madrid

Ingeniería de Sistemas y Automática

Puesta en marcha del sensor fuerza/par JR3

Proyecto realizado por:

Carlos de la Hoz Najarro

Tutor:

Santiago Martínez de la Casa

A mis padres, Salvadora y Jesús, sin vosotros nunca hubiese llegado a ser Ingeniero.
Muchas gracias.

A mi hermana y mi hermano, Patri y Manu.

A mi familia.

Cuando tengas que elegir entre dos caminos,

pregúntate cuál de ellos tiene corazón.

Quien elige el camino del corazón,

no se equivoca nunca.

Popol _Vuh

Las tres leyes de la robótica:

1. Ningún robot causará daño a un ser humano o permitirá, con su inacción, que un ser humano sufra algún mal.

2. Todo robot obedecerá las órdenes recibidas de los seres humanos, excepto cuando esas órdenes puedan entrar en contradicción con la primera ley

3. Todo robot debe proteger su propia existencia, siempre y cuando esta protección no entre en contradicción con la primera y segunda ley.

Isaac Asimov

… y pasaron 9 años

Agradecimientos

Le dedico este Proyecto Fin de Carrera a una Enseñanza Pública, Universal, Plural, gratuita, de todos para todos.

Como consecuencia de un sistema vergonzoso, triste, ineficaz, estamos inmersos en una tradición en la que, al igual que las ganancias se las reparten unos pocos, las pérdidas se socializan, recortes en educación pública, recortes en investigación, recortes en desarrollo, recortes en sanidad, recortes en derechos fundamentales. Todo lo que conseguimos con sangre, sudor y lágrimas, lo perdemos, involucionamos.

Por esto, con todo mi corazón y con toda mi rabia, le agradezco a mi queridísimo Colegio Público Doctor Federico Rubio, a mi apreciado IES Joaquín Turina, por supuesto público, y a mi admirada Universidad Pública Carlos III de Madrid, el que yo, de familia obrera, pudiese desarrollar todo mi potencial intelectual para poder llegar a ser Ingeniero, estudios que de otra forma nunca hubiere podido realizar.

Muchas gracias a mis padres, Salvadora y Jesús. Muchas gracias a mis hermanos, Patri y Manu. Muchas gracias al resto de mi familia, tíos y primos.

Le agradezco a mi tutor, Santiago Martínez de la Casa, su ayuda siempre que la he necesitado.

Gracias a Rafael Heras Martín, a Saúl Herranz Moreiro, a Fernando Sáez Navarro, por su ayuda durante todos estos años.

Gracias a Daniel Morales Tejero, por su gran ayuda con las funciones MEX.

Y también, muchas gracias a Mario Prats, Doctor Ingeniero Informático de la Universidad Jaume I de Valencia, porque cuando estaba inmerso en la más profunda penumbra, ya desesperado por una estéril investigación sobre los drivers necesarios para el funcionamiento de la PCI a utilizar en este trabajo, él apareció y me hizo la luz con su gran generosidad, me permitió el uso de los drivers realizados por él mismo. Eso es lo bonito, que entre todos nos ayudemos, la fluidez de ideas para que con lo mucho que haga uno, otros puedan hacer otro poquito y así desarrollar nuevos proyectos, nuevas

aplicaciones, nuevas tecnologías. Porque nunca olvidemos que la riqueza de un pueblo es su riqueza cultural, es su riqueza intelectual, es su riqueza educativa.

Resumen

Este estudio se engloba dentro de los proyectos realizados por el grupo de trabajo *Robotic Lab* de la *Universidad Carlos III de Madrid* para el desarrollo del nuevo Humanoide RH-2.

Concretamente, implementaremos sobre software libre (Linux) una aplicación C++ que nos permita obtener de la tarjeta de adquisición de datos de 4 puertos PCI P/N 1593 de JR3 inc., en tiempo real, los valores de fuerza/par del sensor industrial *Force Moment Sensor 85M35A3-I40-DH12,* de la compañía JR3 inc., que llevará el humanoide RH-2 instalados en cada uno de sus tobillos. Hay que hacer notar que dicha tarjeta PCI de la empresa JR3 inc. es compatible con cualquier sensores de las series fuerza/par proporcionadas por dicha compañía.

Con este diseño de adquisición de datos tendremos las fuerzas y momentos que ejerce continuamente el robot en los tres ejes de coordenadas xyz sobre sus tobillos, información que, junta a otros parámetros, nos permitirá controlar el equilibrio del humanoide. El objetivo es poder trabajar sobre todos estos datos, de ahí que realizaremos una interfaz gráfica para Matlab, la cual nos permitirá: visualizar en tiempo real los valores de fuerza/par generados por el robot en su caminata, la representación de dichos valores respecto al tiempo, la representación tridimensional del vector fuerza resultante y el cálculo y representación de la trayectoria del ZMP (punto de momento cero) en apoyo simple. Para trabajar en Matlab con datos obtenidos mediante lenguaje C++ haremos uso de las funciones MEX que permiten ejecutar en MATLAB programas implementados en C++.

En consecuencia, el sensor de fuerza/par permitirá al robot tener la información esencial para poder calcular el ZMP (punto de momento cero) y con él poder modificar en tiempo real (a través del sistema de control fuera del estudio de este trabajo) la posición de sus elementos si detecta que dicho punto se encuentra fuera al límite de la denominada área efectiva de estabilidad.

Palabras Clave: Humanoide RH-2, sensor fuerza/par JR3, tarjeta PCI, adquisición y representación datos, Matlab, C++, mex function, interfaz gráfica, control y programación de robots.

Índice

Anexos

Indice de figuras

Indice de tablas

Indice de gráficas

1. Introducción y objetivos

1.1 Introducción

La robótica es consecuencia lógica de la automatización de los sistemas productivos nacidos con la revolución industrial, buscando la máxima eficiencia de los mismos. Su característica fundamental, que lo diferencia de las simples máquinas automáticas, es la versatilidad, esto es, la capacidad de ser adaptado para diversas funciones a través de simples cambios en su programación.

Su desarrollo actual se inició en 1954 con la aparición del brazo articulado diseñado por G. Deval, que posteriormente fue modificado y considerablemente mejorado por V. Scheinman en 1975, al desarrollar el PUMA (brazo manipulador universal programable). El PUMA (Figura 1.1) se considera el precursor de los actuales robots [1].

Figura 1.1: Robot PUMA 500

Los robots han ido evolucionando al ritmo que lo han hecho sus sistemas de control. De los primeros robots con control basados en paradas fijas mecánicas, pasando por el control mediante cintas perforadas y magnéticas, hemos llegado a una tercera generación de robots que requieren lenguajes específicos de programación y cuyo manejo se sustenta en el uso de ordenadores y una serie de sensores que aportan información sobre el medio de trabajo, con lo que pueden variar sus estrategia de trabajo y control.

Toda esta evolución de la robótica responde a la necesidad del hombre para evitar realizar tareas peligrosas, pesadas, rutinarias (cadenas de montaje) o que requieran de una gran precisión (industria informática, cirugía láser y telecirugía). [1]

Figura 1.2. Robot quirúrgico Da Vinci de la empresa Intuitive Surgical.

En la actualidad se está dando un paso más allá al encontramos en pleno desarrollo de la cuarta generación de robots. Ésta se diferencia de la tercera por poseer más y mejores sensores y por utilizar la lógica difusa para su programación de acuerdo con modelos de actuación específicamente programados, lo que sugiere que poseen una cierta capacidad de decisión [1]. Dentro de esta generación se encuentran los

humanoides, robots con forma humana, siendo un referente el robot de ASIMO de Honda (Figura 1.3), que es considerado desde 2005 el robot humanoide más avanzado hasta la fecha.

Figura 1.3. Humanoide ASIMO de Honda.

En este marco de perfeccionamiento de la cuarta generación de robots se encuentra los trabajos realizados, y por ende este Proyecto Fin de Carrera, por *el Departamento de Ingeniería de Automática y Sistemas* de la *Universidad Carlos III de Madrid* para el desarrollo del Humanoide RH-2, sucesor del RH-0 y RH-1.

El RH-2 pertenece a los llamados robots inteligentes capaces de ejecutar funciones diversas controladas por ordenador, al tiempo que puede relacionarse con su entorno y tomar cierto tipo de decisiones sin la intervención directa del hombre.

Uno de los objetivos de estos trabajos es conseguir que el humanoide ande, suba/baje escaleras, de forma segura sin que pierda el equilibrio. Para ello el robot deberá poseer un sistema de control realimentado, entre otros factores, con los datos proporcionados por dos sensores fuerza/par que llevará instalados en sus tobillos. A

continuación podemos observar un diagrama de bloques que muestra el sistema de control (Figura 1.4, en rojo y azul objetivos de nuestro proyecto).

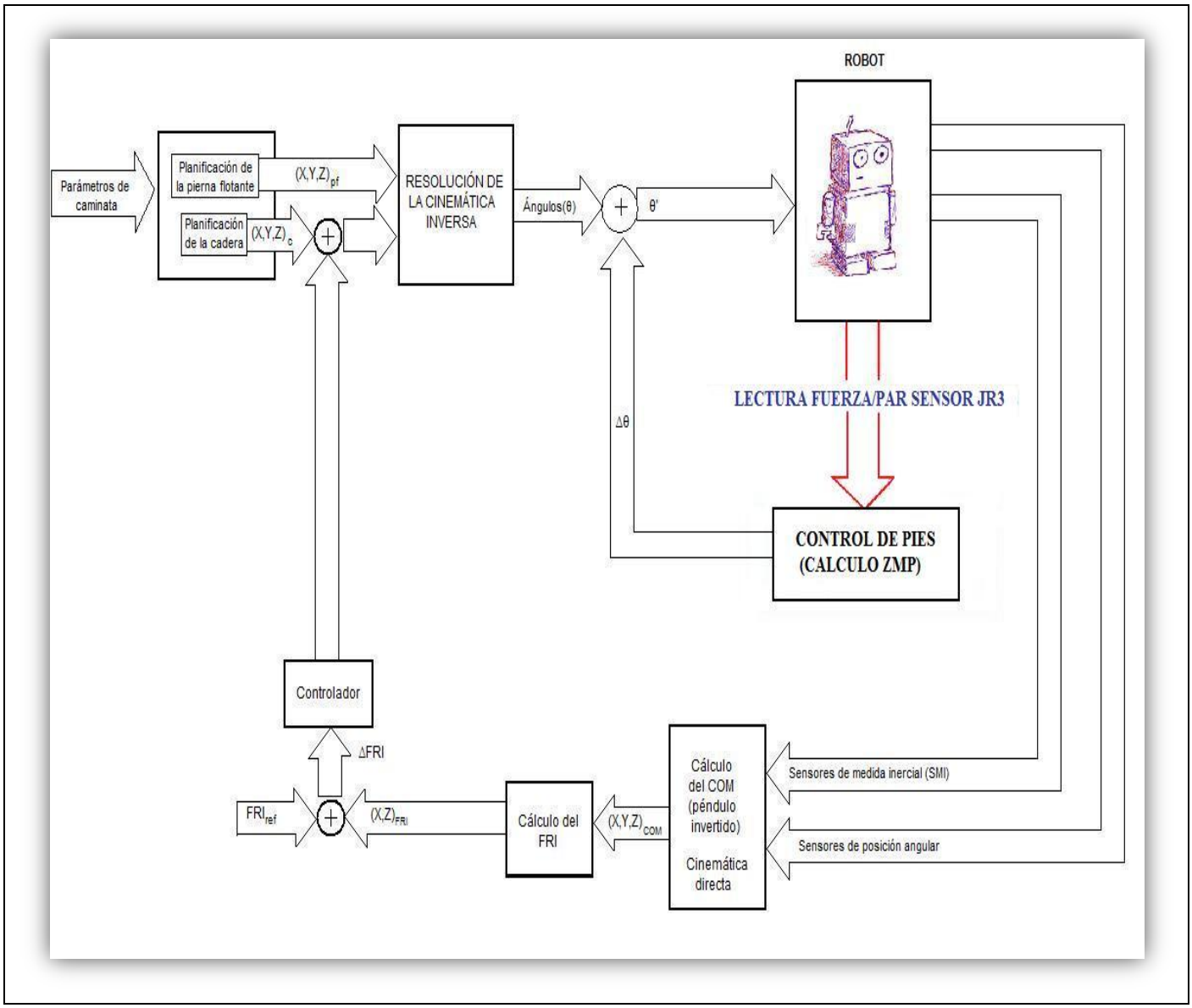


Figura 1.4: Sistema de control de la caminata del robot en lazo cerrado [2].

Podemos observar en la Figura 1.4 como el conocimiento de las fuerzas y momentos que realiza el humanoide sobre sus tobillos radica en la necesidad de calcular el ZMP (punto de momento cero) y es que el ZMP es un concepto muy importante para

el equilibrio de un robot durante sus movimientos. En el apartado 2.5 del proyecto explicaremos con detenimiento el criterio de estabilidad basado en el ZMP.

En resumen, con el fin de controlar la caminata del robot es necesaria la caracterización de un sensor fuerza/par. Nuestra elección ha sido el sensor industrial Force Moment Sensor 85M35A3-I40-DH12, de la compañía JR3 Inc. Dicha caracterización la haremos en sistema operativo Linux. Diseñaremos una aplicación C++ que, en tiempo real, guarde en un fichero las fuerzas y momentos producidos por el robot en los tres ejes de coordenadas. Para la visualización y representación gráfica de estos datos diseñaremos una interfaz gráfica mediante MATLAB. Interfaz con la que también calcularemos y representaremos la trayectoria del mencionado punto de momento cero. Para la instalación y activación de la tarjeta utilizamos los driver diseñados por el Doctor Ingeniero Informático Mario Prats.

Fuera del alcance de este proyecto, y como continuación y complementación del mismo, se deberá implementar un sistema de control, semejante al mostrado en la Figura 1.4, que al detectar a nuestro ZMP fuera del área de estabilidad (el denominado polígono de soporte), realimente el sistema con la información necesaria para que los motores rectifiquen el movimiento del RH2 con el fin de mantener al ZMP dentro del polígono de soporte y, en consecuencia, impedir así la caída del humanoide.

Todo el desarrollo de este proyecto se ha llevado a cabo utilizando el péndulo simple invertido de 2 GDL y el robot ABB con los que cuenta el *Departamento de Automática y Sistemas* de la *Universidad Carlos III de Madrid* en su Escuela Politécnica Superior del Campus de Leganés. El péndulo invertido simula el tobillo del robot humanoide RH-2 y posee instalado en su base el sensor a caracterizar: Sensor fuerza/par 85M35A3-I40-DH12, de la compañía JR3 inc. Del mismo modo, se ha utilizado un PC con sistema operativo Linux (kernel 2-6.26-2-686), compilador gcc y Matlab, en el que se ha instalado la tarjeta de adquisición de datos PCI P/N 1593 de JR3 inc. El PC se ha comunicado con el sensor JR3 del péndulo invertido mediante un cable modular RJ-11 (6 pines) conectado en el otro extremo a uno de los 4 puertos de la tarjeta PCI.

1.2 Objetivos

Como ya hemos comentado, tenemos tres objetivos a conseguir con este proyecto. Trabajando con sistema operativos Linux, buscamos:

1º). Diseñar una aplicación en C++ que, en tiempo real, obtenga los valores en Sistema Internacional de fuerza y par que lee la tarjeta PCI del sensor JR3.

2º). Diseñar una interfaz gráfica en MATLAB que:

- Visualice, en tiempo real, los valores de fuerza/par generados por el robot sobre sus apoyos,
- Represente gráficamente las fuerzas/pares respecto al tiempo,
- Represente el vector fuerza resultante,
- Calcule, en tiempo real, el ZMP en apoyo simple.
- Y represente la trayectoria del ZMP en apoyo simple.

3º). Diseñar un sistema de comunicación entre la aplicación C++ y la interfaz gráfica. Para ello se creará una zona de memoria compartida con la que MATLAB se comunicará a través de funciones MEX.

Estos 3 objetivos forma parte de un objetivo global dentro del desarrollo del Humanoide RH-2 a alcanzar junto a otros trabajos, que es la implementación adecuada de un sistema de control que realimente, entre otros factores, la información del ZMP calculado en cada instante, con el fin que el humanoide camine de forma segura sin perder el equilibrio.

1.3. Estructura de la memoria

La memoria del proyecto se estructura en cinco capítulos que a continuación se describen brevemente.

- El **capítulo 1** comienza con una pequeña introducción al proyecto para dar una visión global de lo que va a tratar y de los principales objetivos a alcanzar.

- En el **capítulo 2** contextualizaremos el proyecto describiendo los humanoides RH-0 y RH-1. Después realizaremos una descripción general del humanoide RH-2, dentro de la cual haremos un estudio general de los diferentes sensores de fuerza/par existentes en el mercado. Los dos últimos apartados los dedicaremos a explicar el criterio de estabilidad basado en el ZMP y la teoría del equilibrio bípedo, conceptos necesarios para la correcta caminata de un humanoide.

- El **capítulo 3** versará ya específicamente del tobillo del robot, elemento en el que se centra nuestro proyecto. En el primer apartado hablaremos de los elementos que constituyen el tobillo para después concretar las especificaciones de los elementos sobre los que trabajamos directamente: sensor fuerza/par JR3, tarjeta de adquisición de datos de 4 sensores PCI P/N 1593 de JR3 inc. y cable RJ-11 utilizado para la conexión sensor-PCI.

- En el **4º capítulo** explicaremos cómo funcionan las diferentes aplicaciones diseñadas en C++ y MATLAB. Al final del capítulo mostraremos y comentaremos las tablas de datos tomadas en las medidas experimentales, así como, sus representaciones gráficas.

- En el **capítulo 5** se muestra el presupuesto del proyecto.

- En el **capítulo 6** se encuentran las conclusiones a las que se han llegado y las posibles mejoras propuestas.

- Al final del trabajo se incluye la bibliografía con las principales fuentes de información consultadas.

- Anexado al proyecto están los códigos de las aplicaciones diseñadas, hojas de características y planos de los componentes utilizados.

2. Estado del arte

El proyecto de robots humanoides RH de la *Universidad Carlos III de Madrid* consta de dos prototipos ya construidos: RH-0 y el RH-1. El RH-1 suponía simplemente la modificación y sustitución de algunos elementos hardware y rediseños mecánicos del RH-0. Así, a excepción de algunos cambios estructurales, se mantuvo el tamaño y los 21 GDL con los que ya contaba el RH-0.

A diferencia del RH-1, el RH-2 es más ambicioso y busca cerrar el ciclo de control de caminata con toda la información que requiere para que el robot mantenga el equilibrio en todo instante, ya que, ni el RH-0 ni el RH-1, poseen un sistema de medición de error de la posición de las articulaciones y tampoco conocen las fuerzas y momentos que ejercen sobre el suelo, información esencial para una correcta caminata. Con el fin de solucionar estos dos problemas, por un lado se implementará los sensores de sincronismo para realizar el homing y buscar la posición de referencia de los motores para, al compararla con la posición real leída a la salida de las transmisiones, conocer el error de posicionamiento de las articulaciones. Para ello, será necesario introducir un encoder más en cada transmisión y diseñar el circuito de control y adquisición de datos. Por otro lado, se instalará sensores fuerza/par en los tobillos del RH-2 para conocer en todo momento las fuerzas y pares ejercidos por los pies sobre el suelo en los tres ejes. Como ya se sabe, la implementación y caracterización de estos sensores es el objetivo de nuestro proyecto. Estas mejoras añadirán al RH-2 tres GDL más respecto a sus antecesores, presentando un total de 24 GDL.

2.1. Humanoides RH-0 y RH-1

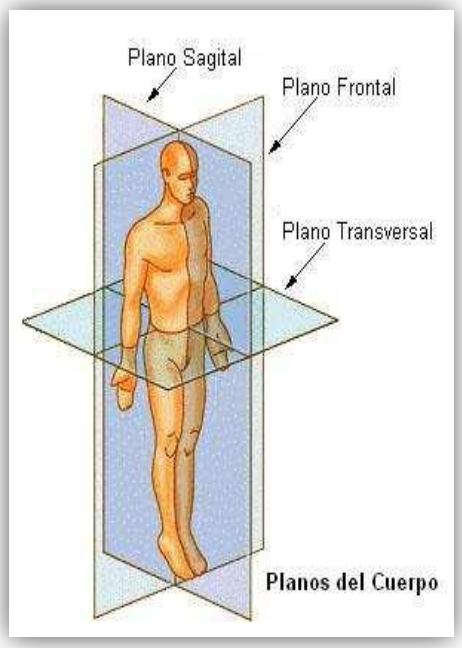
El RH-0 y RH-1 están diseñados para:

1) Caminar en línea recta. El RH-1 alcanza los 0,8 km/h.
2) Subir y bajar escaleras.
3) Simular al caminar el movimiento de brazos de una persona.
4) Gesticular con los brazos: Saludar, señalar…

Con estas especificaciones de funciones, el diseño mecánico y software de los antecesores al RH-2 es el que se describe en los dos siguientes apartados.

2.1.1. Diseño mecánico

Tanto el RH-0 como el RH-1, presentan 21 GDL. La distribución de estos grados de libertad se muestra en la Tabla 2.1.



		Nº GDL	Eje movimiento
Piernas (12 GDL)	Cadera	3(x2)	Sagital, Frontal Transversal
	Rodillas	1(x2)	Sagital
	Tobillos	**2(x2)**	**Sagital Frontal**
Brazos (8 GDL)	Hombros	2(X2)	Sagital Frontal
	Codos	1(X2)	Sagital
	Muñecas	1(X2)	Transversal
Tronco (1 GDL)	Tronco	1	Transversal
Cabeza (2GDL)	Cámara	2	Transversal(izquierda-derecha) Sagital (Arriba-Abajo)
TOTAL	**21 GDL (23 con la cámara)**		

Tabla 2.1. Distribución de GDL de los humanoides RH-0 y RH-1 [3].

Las dimensiones del RH-0 son:

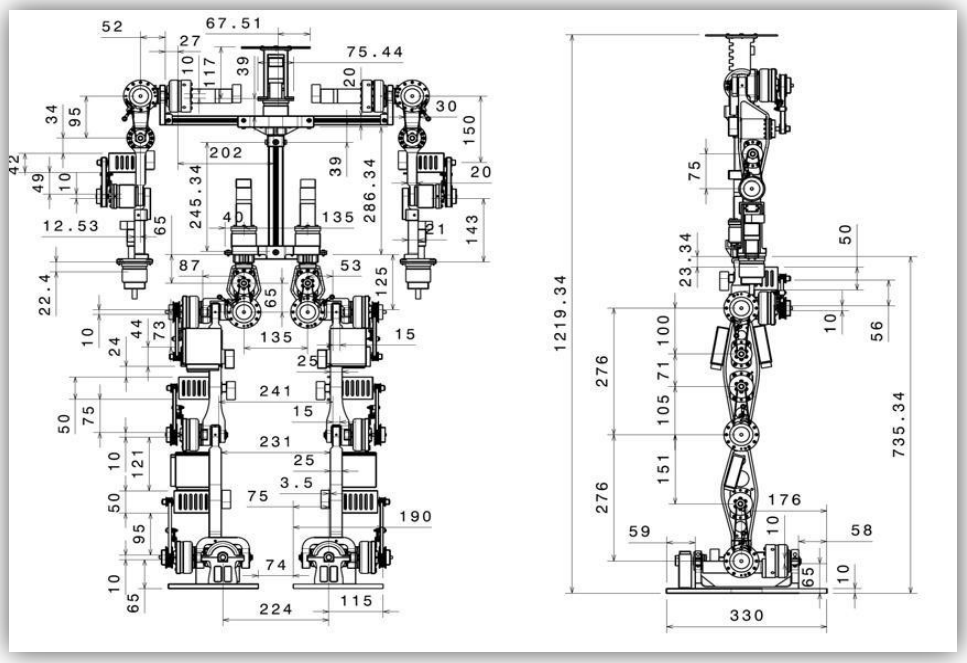


Figura 2.1. Diseño mecánico humanoide RH-0.

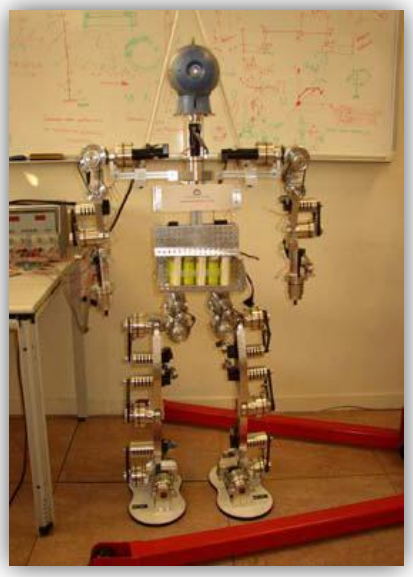


Figura 2.2. Humanoide RH-0.

Figura 2.3. Humanoide RH-1.

Como se puede comprobar en la Figura 2.1, la altura del RH-O es de unos 120 cm. Esta altura la supera el RH-1 al alcanzar los 150 cm. En la Figura 2.2 y Figura 2.3 se muestra, respectivamente, el RH-0 y el RH-1, ambos ya construidos.

2.1.2. Diseño del sistema hardware y software.

La arquitectura software del RH-0 y RH-1 se basa en tres capas como se muestra en la Figura 2.4:

Figura 2.4. Distribución del software en capas [3].

Las diferentes capas están interconectadas a través de comunicaciones inalámbricas (capas de organización y comunicación) y del bus del campo CANbus (capas de coordinación y ejecución).

La Tabla 2.2 refleja las variaciones en los elementos utilizados entre el RH-0 y el RH-1.

		RH-0	RH-1
ACCIONADORES	**Motor**	Faulhaber 24 V DC	Faulhaber 24 V DC
	Frenos motor	MBZ 22 de Faulhaber	MBZ 22 de Faulhaber
SENSORIZACION	**Encóder**	HEDS 5540 A de Faulhaber	HEDS 5540 A de Faulhaber
	Giróscopo	Silicon CRS07	Silicon CRS07
	Inclinómetro	ADXL Analog Devices	ADXL Analog Devices
	Sincronismo	contrinex serie 620	contrinex serie 620
COMPUTACION	**Microprocedador**	SECO M570 PC/104 400MHz	Pentium M PC/104 Digital Logic
	Driver	Elmo HARMONICA A5/50 CAN	Elmo HARMONICA A5/50 CAN

Tabla 2.2.Comparativa de los elementos entre los humanoides RH-0 y RH-1 [3].

Como se puede observar en la Tabla 2.2, las diferencias de elementos entre los dos primeros robot de la serie RH es mínima, únicamente difieren en el PC utilizado. Las ventajas del PC del RH-1 son:

- Mayor capacidad.
- Bus más avanzado (PC\104+) lo que supone mayor velocidad.
- Además, el nuevo controlador puede trabajar en modo de interrupciones.

Común a ambos sistemas hardware es el bus interno ISA dedicado al tratamiento digital de imagen y sonido, y otro igual para el control de la red CAN y el control de las trayectorias de las articulaciones en función de los datos sensoriales. Para determinar la posición angular de los motores, se obtiene la información directamente del encóder relativo [2].

Tras esta descripción conjunta de los dos primeros humanoides de la saga RH, ahora vamos a realizar un corolario de las características del RH-1, modelo que tomará como punto de partida nuestro RH-2. Así, tal como nos explica la Doctora Ingeniera Industrial Concepción Monje Micharet de la Universidad Carlos III de Madrid, en su trabajo "Control de la caminata del robot humanoide RH-1", el robot humanoide RH-1:

- Mide 150 cm, pesa 50 kg y cuenta con 21 GDL.

- Puede andar hasta 0.8Km/h en línea recta, rotar y andar lateralmente, recibir comandos de voz, emitir respuestas, reconocer y contestar comandos visuales mediante gestos, reconocer caras y localizar la posición y orientación de un objetivo

- Consta de hardware totalmente embebido y baterías con una autonomía de 30 minutos.

- Posee dos ordenadores principales en formato PC-104, uno para comandar los motores y otro para el procesamiento de imágenes y sonidos.

- Dado que el control de los motores es distribuido, dispone de un servodriver para cada motor.

- Los motores se comunican con el PC-104 respectivo mediante un bus de campo (CANBUS).

- Para el control de la estabilidad del robot, dispone de sensores inerciales (acelerómetros y giróscopos) que permiten compensar los efectos gravitatorios e inerciales del robot mientras se está moviendo. Para comunicarse con los inclinómetros y giróscopos se recurre a una conexión serie RS-232.

- La cámara que posee tiene dos grados de libertad (pantilt) que permiten incrementar su rango visual y, junto con los micrófonos y altavoces, son comandados por el otro PC-104.

- Es operado mediante comunicación inalámbrica según protocolo 802.11b desde un ordenador portátil o Workstation.

Como ya se ha comentado con anterioridad, se puede observar en la Tabla 2.2 como ni el RH-0 ni el RH-1 poseen un sensor de fuerza/par en sus tobillos que les permita determinar el ZMP. Por lo tanto, en este trabajo partimos de cero para el desarrollo de una aplicación en sistema operativo Linux que nos permita leer el sensor fuerza/par que sí llevará el RH-2.

2.2. Humanoide RH-2

2.2.1. Diseño mecánico

El RH-2 supera los 150 cm del RH-1alcanzando una altura de 165 cm más acorde con la del ser humano. Su peso será de unos 60 Kg. Se estima que podrá transportar objetos de 2 Kg de peso e incluso sentarse. Las medidas del humanoide RH-2 se muestran en la Figura 2.5.

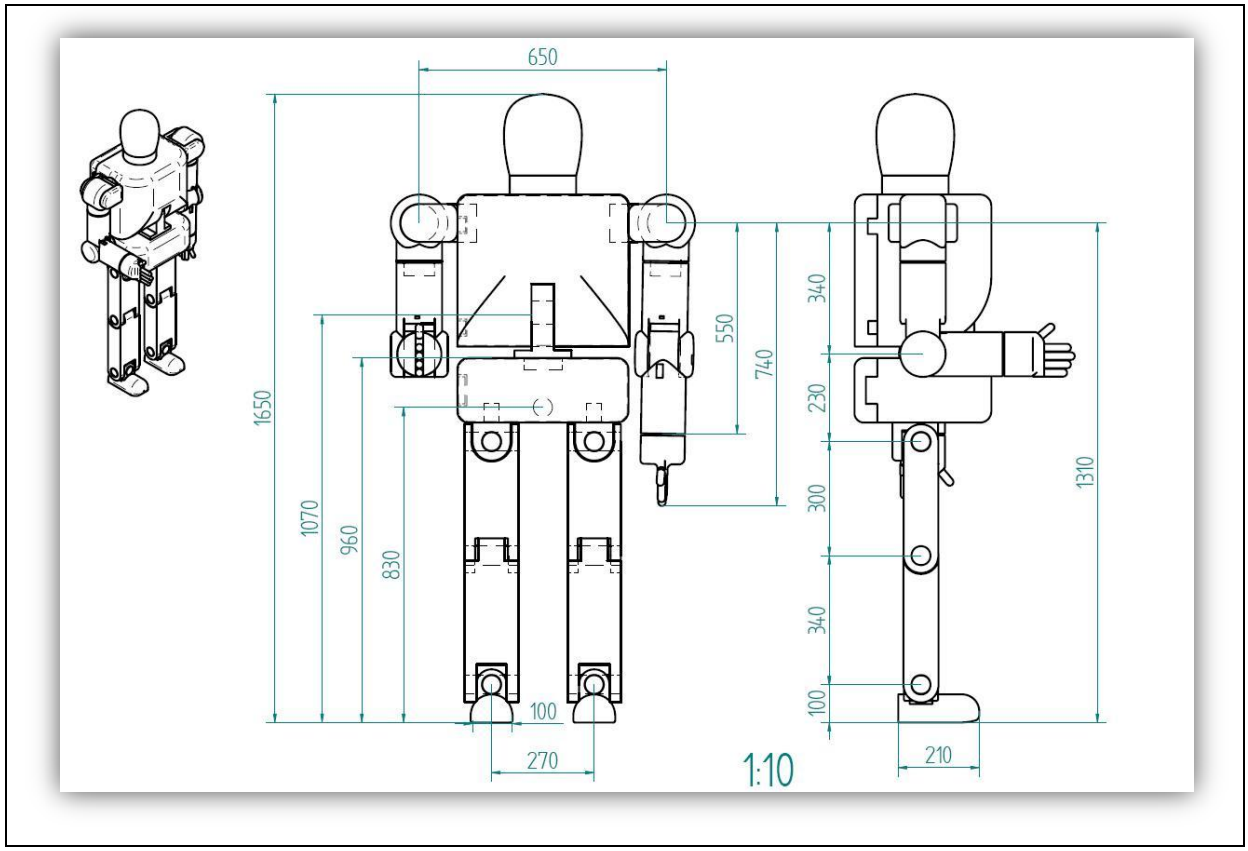


Figura 2.5. Medidas humanoide RH- 2.

El nuevo modelo, RH-2 posee 24 GDL (26 si se tienen en cuenta los correspondientes a los motores de la cabeza). Tres grados de libertad más respecto a los anteriores humanoides de la saga RH, distribuidos como sigue:

- Uno en cada codo en el plano transversal, que permita al brazo realizar movimientos más parecidos a los humanos.

- Y un tercero en el plano frontal del tronco que permitirá controlar de manera más rápida el balanceo hacia delante y hacia atrás del cuerpo y lograr mantener su centro de masa en una posición correcta. Este último grado de libertad dota al robot de la capacidad de plegar más el tronco y poder sentarse.

La distribución por articulación de todos estos GDL es:

- **Piernas**: Dispone cada una de 6 GDL distribuidos entre el tobillo, la rodilla y la cadera. La cadera posee 3 de ellos, uno en el plano sagital, otro en el frontal y el tercero en el plano transversal, utilizado en el cambio de dirección de movimiento. La rodilla tiene 1 GDL en el plano sagital. Y por último el tobillo posee 2 GDL, uno en el plano sagital para adaptar el pie al suelo, y otro en el plano frontal que permite el balanceo para mantener el equilibrio.

- **Brazos**: Cada uno de los brazos dispone de 5 GDL distribuidos entre el hombro, el codo y la muñeca. En el hombro existen 2 GDL en los planos sagital y frontal. En la muñeca existe únicamente 1 GDL en el plano transversal. Esta distribución permitirá manipular objetos.

- **Tronco**: El tronco posee 2 GDL, uno en el plano transversal que le permite el giro en ese plano sin tener que mover las piernas, y otro en el plano frontal que le permite regular su inclinación.

2.2.2. Arquitectura Hardware

Con el RH-2 se busca una solución, utilizando las tecnologías hoy en día disponibles, para los problemas derivados de las ineficiencias ya descritas de las dos anteriores versiones. Se quiere:

- Que, en la medida de lo posible, el ciclo de control de la caminata del robot se cierre en el mínimo tiempo posible. Es imprescindible que se pueda realizar este lazo de control, ya que el robot debe funcionar en un lazo cerrado para mantener siempre la estabilidad.

- Recurrir a sistemas lo más distribuidos posibles que nos aseguren tiempos de cómputo más reducidos.

- Se debe demostrar la viabilidad del protocolo CAN-bus utilizado en los anteriores robots y comprobar que el hardware que gestiona esta red es el más apropiado.

- Los microprocesadores usados anteriormente se basaban en el protocolo ISA para la comunicación interna. Estos dispositivos fueron problemáticos y ahora se estudiarán distintos protocolos interno, en el del sensor fuerza/par comunicación mediante tarjeta PCI, y se buscarán otras opciones de microprocesadores.

- La información proporcionada por los sensores debe actualizarse en tiempo real. Por ello, las consultas entre los microprocesadores que gestionan los sensores debe ser lo más rápidas posibles. Se han de estudiar las distintas posibilidades de comunicación inalámbrica en busca de un acceso eficaz al sistema que permita el control del robot.

- Elementos ligeros y compactos. En concreto, el hardware necesario para el control de los motores que gestionaban los 21GDL del RH-1 era muy voluminoso. Ahora, en el RH-2, nos encontramos con 3 grados más, lo

que implica mayores números de elementos cuando lo que se pretende es un robot más ligero y menos voluminoso. De ahí que se requieran elementos ligeros, pequeños, de bajo consumo y que cumplan o mejoren las especificaciones de los utilizados en el modelo RH-1.

En busca de satisfacer los anteriores requisitos, Gonzalo Funes Romero, Ingeniero Técnico de Electrónica Industrial por la Universidad Carlos III de Madrid, nos propone en su Proyecto Fin de Carrera "Diseño e implementación de los sistemas electrónicos del tren inferior del robot humanoide RH-2", la siguiente arquitectura hardware para el RH-2:

Se basa en dos microprocesadores como sistema computacional, uno que envía órdenes a las piernas (caminante), y otro a los brazos (manipulador).

- El procesador principal (caminante) se encargará de procesar la información de los sensores para lograr mantener el equilibrio y, en consecuencia, una caminata estable. Tendrá dos tarjetas conectadas. Una con protocolo CANopen y que puede gestionará dos redes Can bus a la vez, cada una de las cuales controlará los motores de una pierna. La otra tarjeta PCI, sirve de interfaz entre el microprocesador y los sensores fuerza/par.

- Por su parte, el procesador secundario (manipulador) se encargará de controlar el movimiento de los brazos. También será el encargado de controlar y recibir información del entorno, para una vez procesados estos datos actuar en consecuencia y llevar a cabo tareas de manipulación de objetos, evitar obstáculos, etc. Al igual que el procesador principal, dispondrá de dos tarjetas PCI. La función de la primera de ellas será controlar los motores de los brazos y el tronco a través de las redes Can bus 3 y 4. La otra tarjeta PCI se utilizará para comunicar los sensores fuerza/par con el microprocesador.

Con esta arquitectura se quiere conseguir un sistema distribuido entre dos microprocesadores principales que, a su vez, estén conectados con los microprocesadores de los drivers y encóders. El objetivo de esta división del sistema es obtener un mayor rendimiento y una mayor velocidad de cálculo para cada una de las partes [2].

En cuanto a las comunicaciones, se añadirá un router que creará una red entre los microprocesadores y habilitará posibles conexiones con el exterior, a través de wifi [2].

En la Figura 2.6 se muestra un esquema de la arquitectura hardware propuesta.

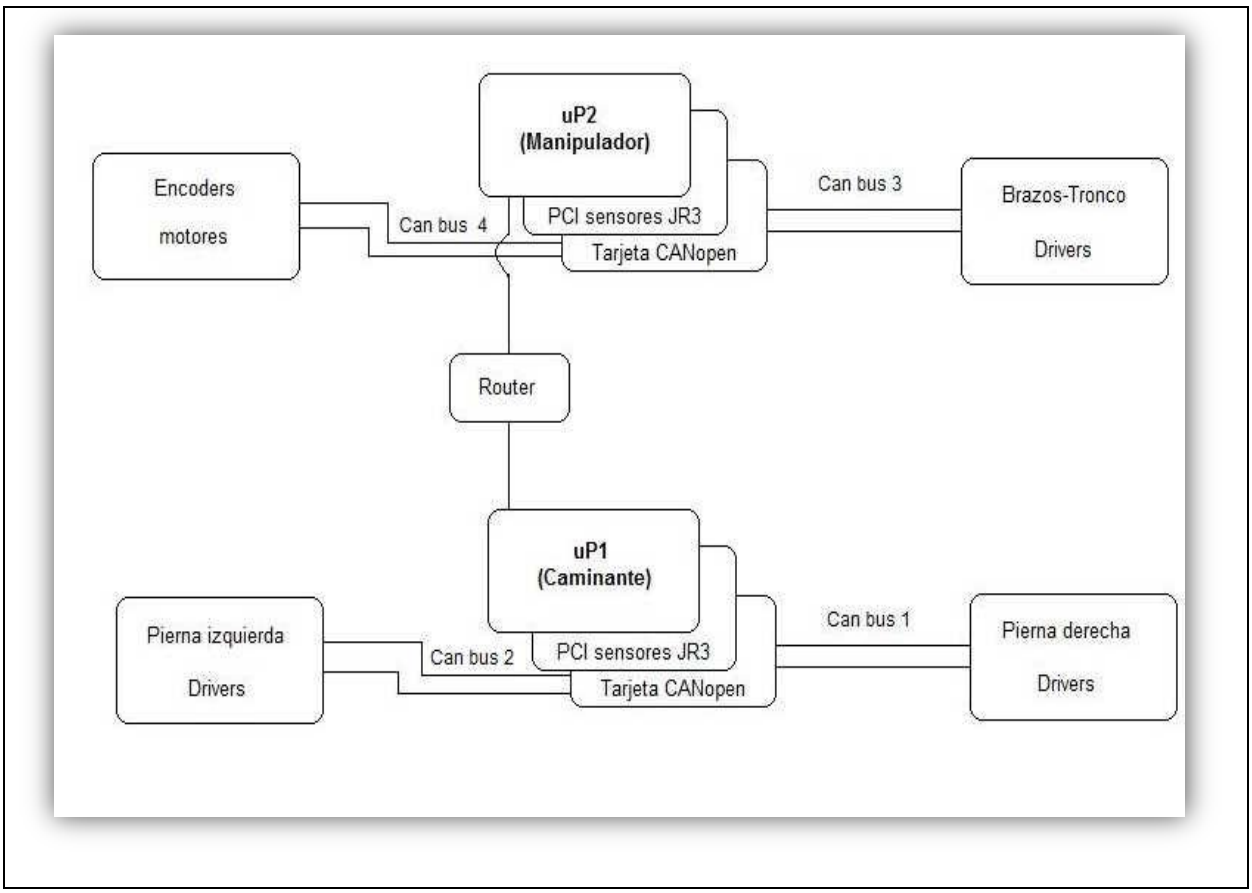


Figura 2.6. Arquitectura Hardware RH-2 [2].

Por último, como ya comentamos en la introducción a este capítulo, en el RH-2 se implementa dos funciones no desarrolladas en sus antecesores, que son:

- Diseño de un sistema de medición del error de la posición de las articulaciones para un correcto posicionamiento de las articulaciones en el movimiento del RH-2
- Medición de la fuerza/par ejercidos por el humanoide sobre sus tobillos con el fin de calcular el ZMP, cuyo conocimiento es necesario para que el robot se mantenga en equilibrio durante la caminata.

Para el cálculo de error del posicionamiento de las diferentes articulaciones es necesario, como es obvio, conocer la posición de cada una de las articulaciones, para lo que se ha de medir la posición angular. Esta posición ha de obtenerse en términos absolutos con el fin de conocer donde están realmente los eslabones y poder generar, en consecuencia, las trayectorias correctamente. Conociendo la posición absoluta, no es necesario un sistema de sincronismo como el que utilizaba el RH-1.

El sistema de medición de las posiciones de las articulaciones se muestra en la Figura 2.7:

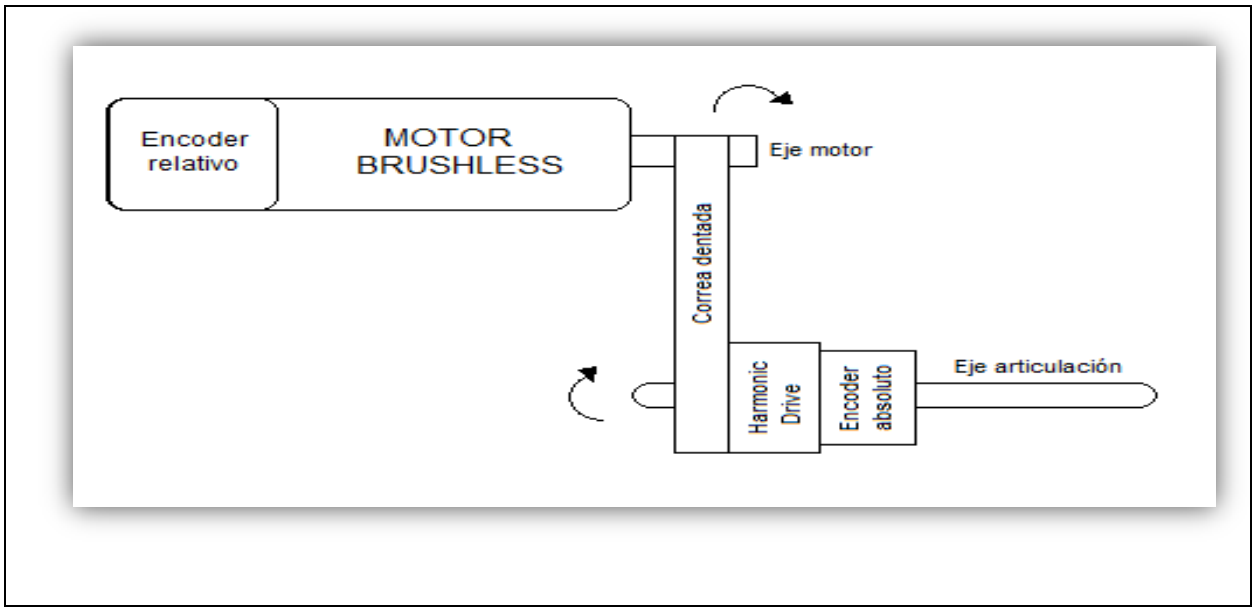


Figura 2.7. Sistema de medición para las posiciones articulares.

Como se observa en la Figura 2.7, se utilizarán encóders absolutos para medir la posición absoluta de su motor asociado. Si se conoce la posición del motor, se puede conocer la posición absoluta de la articulación. Estos encóders absolutos irán colocados en el eje de la articulación, donde pueden medir la posición angular. Se comunicarán con su microprocesador correspondiente a través de un PIC y un receptor-transmisor. Los motores a utilizar serán motores brushless CC (sin escobillas) de diseño plano que dispondrán de sensores Hall para medir la posición relativa del mismo. A cada motor se le acoplará un encóder relativo en su eje que, funcionando junto con los sensores Hall, será capaz de medir la velocidad del motor. Estos dispositivos serán controlados por un driver que será el encargado de procesar la información recibida. A su vez, cada driver estará comunicado por el CAN Bus con su microprocesador correspondiente [2].

En cuanto al sensor fuerza/par, éstos se pueden basar en distintas propiedades de los materiales, dando lugar a distintos métodos de medición. Así, tenemos sensores basados en las propiedades elásticas, piezoeléctricas, piezorresistivas y capacitivas que pueden presentar los diferentes materiales. Basándose en estas características las compañías especializadas ofrecen una gran variedad de modelos con una gran diversidad de características. En el siguiente apartado realizaremos una recopilación de todos estos modelos que existen en la actualidad, seleccionando aquéllos cuyas propiedades satisfagan los requerimientos de nuestro sistema a implementar. Ya en el siguiente capítulo, dónde se explican los elementos utilizados para el desarrollo experimental del presente proyecto, se hablará detenidamente del sensor seleccionado para el robot RH-2 en estudio, como ya se sabe, se trata del modelo 85M35A3-I40-DH de la compañía JR3 inc.

2.3. Sensores de fuerza/par.

Como su nombre lo indica, estos sensores permiten transformar una fuerza aplicada sobre una superficie conocida, en una magnitud eléctrica proporcional: voltaje, corriente, resistencia, etc. [4]

Los robots requieren de este tipo de sensores para calcular, entre otros parámetros, el ZMP, información que durante la caminata será utilizada para ajustar la posición de los actuadores encargados del equilibrio [4].

2.3.1 Tipos de sensores de fuerza según propiedades materiales

Los sensores de fuerza pueden ser de 4 tipos,

a). <u>Sensores Elásticos</u>. Consisten en un material elástico, como espuma o esponja de tipo conductiva, ubicada entre dos placas aislantes protectoras que tienen una doble funcionalidad, por un lado protegen la espuma y, por otro, sirve de soporte de los contactos eléctricos en forma de puntos que sobresalen de las placas, tal como se observa a continuación en la Figura 2.8.

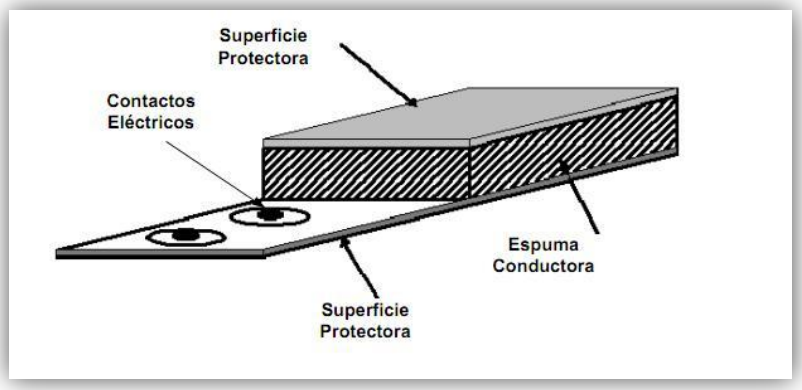


Figura 2.8. Sensor de fuerza elástico.

Cuando una fuerza actúa sobre la superficie protectora, la espuma conductora se deforma, cambiando su densidad en la región deformada, lo cual a su vez varía la resistencia medida entre las dos superficies protectoras. Si bien es un método relativamente fácil de implementar, ya que solo se necesita medir la variación de corriente a través de la espuma, tiene serias desventajas, a saber:

- La variación de la resistencia presenta una curva no lineal excesiva, lo cual la hace difícil de procesar directamente.
- Además, las constantes deformaciones de la espuma, disminuyen considerablemente la vida útil del sensor.

b). Sensores Piezoresistivos: Estos sensores pueden ser considerados como una evolución de los sensores elásticos. Se basan en el principio de la piezoresistividad, fenómeno consistente en la variación de la resistencia de un material al aplicarle una presión sobre sus superficies. Hasta este punto podría ser considerado igual a los sensores elásticos, sin embargo, la diferencia está en que en lugar de colocar una espuma conductora, se coloca un plástico conductor no deformable constituido por partículas conductoras y no conductoras suspendidas en el material. Cuando se aplica una fuerza externa, el material sufre una reordenación de partículas tanto conductoras como no conductoras, lo cual reduce la resistencia del mismo. Este tipo de sensor es uno de los más populares para la medición de fuerza pues no son deformables y por ende tiene una vida útil más prolongada. Además el material piezorresistente presenta una variación más lineal que el de los sensores elásticos.

c). **Sensores Capacitivos**: Para entender de manera más clara como opera este tipo de sensor es necesario recordar brevemente la expresión matemática para la capacitancia de un condensador:

$$C = \varepsilon \cdot A/d$$

Donde “ε” es la permitividad del medio dieléctrico que separa las placas, “A” es la superficie de las placas y “d” es la distancia entre placas del condensador. Si variamos la superficie de las placas o la distancia entre ellas podemos obtener,

respectivamente, una variación directa e inversamente proporcional de la capacidad. En la Figura 2.9 se muestra el esquema de un sensor de fuerza capacitivo que se fundamenta en la variación del área efectiva de un condensador:

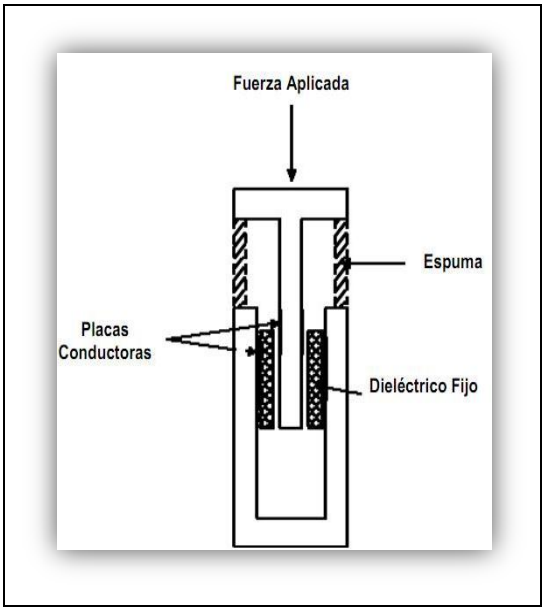


Figura 2.9. Sensor de fuerza capacitivo.

A medida que la fuerza aplicada hunde el eslabón en forma de "T" el área efectiva del condensador va incrementándose, lo que supone un aumento lineal en la capacitancia.

Sus ventajas radican en su bajo consumo de corriente y en poseer una respuesta más lineal que los sensores de tipo resistivo. En contrapartida, posee el inconveniente de requerir un circuito para la conversión de capacitancia a voltaje o corriente. Esta conversión se puede llevar a cabo bien con circuitos de corriente constante, o bien, mediante una transformación de capacitancia a frecuencia, en cuyo caso se utiliza un VCO (Oscilador controlado por voltaje). Todo esto implica circuitos adicionales y, en consecuencia, un mayor costo.

d). <u>Sensores piezoeléctricos</u>: Finalmente, otro de los tipos de sensores más comunes para la medición de fuerza, son los sensores piezoeléctricos, los cuales

generan un voltaje momentáneo proporcional en amplitud a la presión aplicadas. Físicamente un poseen placas protectoras que contienen al material sensor, en este caso dicho material es un cristal de cuarzo o algún tipo de cerámica tratada con propiedades piezoeléctricas. Internamente está constituido de moléculas bipolares ordenadas que actúan como pequeños dipolos eléctricos (cargas eléctricas de polaridad opuesta separadas una distancia específica), que al aplicarles una cierta fuerza producen una pequeña oscilación hasta que se estabilizan, es esta oscilación de cargas lo que genera un voltaje en forma de una sinusoide amortiguada, que posteriormente es procesada por circuitos adicionales. La estructura de este tipo de sensor se observa en la Figura 2.10.

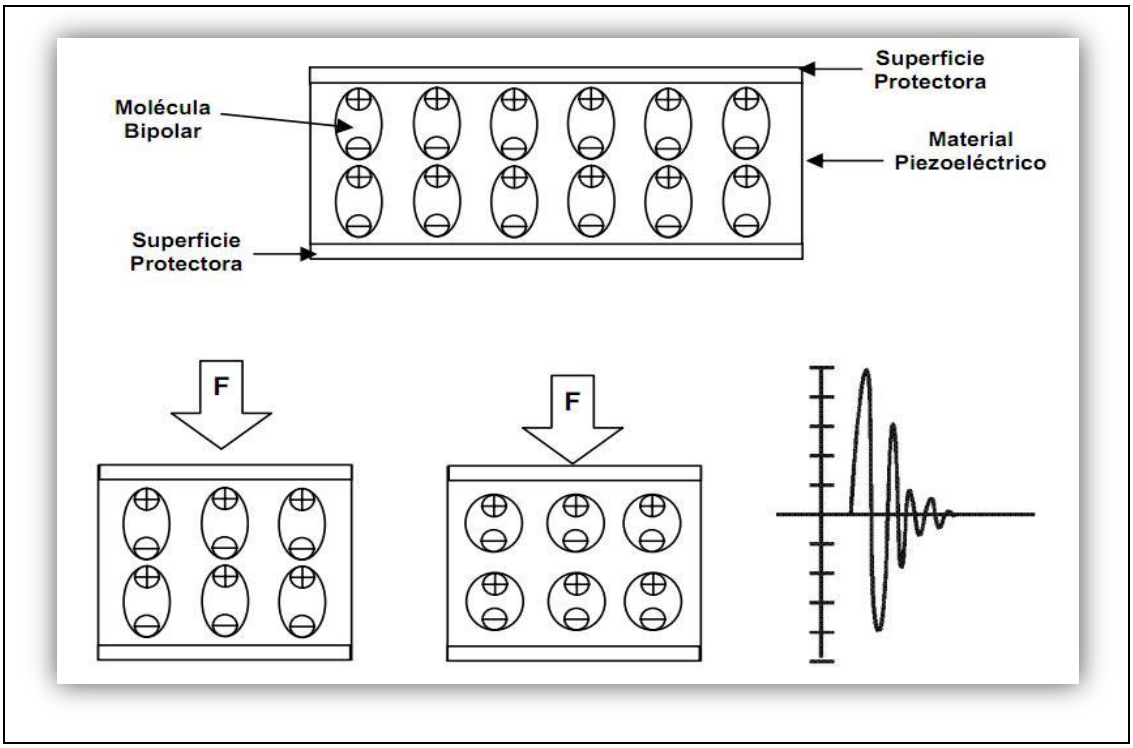


Figura 2.10. Sensor de fuerza piezoeléctrico.

2.3.2. Modelos sensor fuerza/par para aplicaciones robóticas ofertados por las compañías.

El mercado de sensores fuerza/par nos ofrece una amplia gama de sensores para aplicaciones robóticas con un conjunto amplio de rangos tanto en fuerza como en par. Todas las empresas definen, para todos los sensores, al plano XY como el propio del cuerpo del sensor, siendo el eje "z" el eje vertical (Anexo 5).

Nuestro requerimiento es que el sensor sea capaz de medir como mínimo los 65 kg de masa que tendrá el RH-2. Este peso máximo proporciona una fuerza mínima en torno a los 650 N, la cual se proyectará únicamente en un sólo eje (el eje "z" correspondiente a la dirección de la pierna del robot) cuando el humanoide esté totalmente erguido, es decir, en cualquier otra posición de inclinación respecto a la vertical, la fuerza total se descompondrá en valores inferiores a 650N en cada uno de los tres ejes. Por tanto, nuestro valor máximo a medir será de 650N en el eje "z" para dicha posición en vertical. Las empresas proporcionan sensores cuyo rango en el eje "z" es, como mínimo, el doble al rango correspondiente en sus ejes "x" e "y", nos interesan aquello sensores cuyo rango en los ejes "x" e "y" sea superior a la mitad de 650 N, unos 325 N. Sobredimensionando a un valor lógico, buscaremos sensores que nos proporcionen un valor máximo de medida de entre estos 650 N y 1000 N en el eje "z".

Tanto la empresa "ASI Industrial automation" como la "Schunk" ofertan los mismos modelos de sensores, cuyas características básicas se presentan en la Tabla 2.3. ([5], [6]).

Modelo	Max Fx,Fy (N)	Max Tx,Ty (N*m)	Peso (kg)	Diámetro (mm)	Altura (mm)
Nano17 Titanium	**±32**	**±0.2**	**0.00907**	**17**	**14**
Nano43	**±36**	**±0.5**	**0.0408**	**43**	**11**
Nano17	**±50**	**±0.5**	**0.00907**	**17**	**14**
Nano17 IP65/IP68	**±50**	**±0.5**	**0.0408**	**20**	**22**
Mini40	**±80**	**±4 4**	**0.0499**	**40**	**12**
Gamma IP60	**±130**	**±10**	**0.467**	**99**	**41**
Gamma	**±130**	**±10**	**0.254**	**75**	**33**
Gamma IP65/IP68	**±130**	**±10**	**1.09**	**110**	**52**
Nano25	**±250**	**±6**	**0.0635**	**25**	**22**
Nano25 IP65/IP68	**±250**	**±6**	**0.136**	**28**	**27**
Mini45	**±580**	**±20**	**0.0907**	**45**	**16**
Delta	**±660**	**±60**	**0.912**	**94**	**33**
Delta IP60	**±660**	**±60**	**1.81**	**130**	**47**
Delta IP65/IP68	**±660**	**±60**	**2.63**	**100**	**52**
Mini85	**±1900**	**±80**	**0.635**	**85**	**30**
Omega85	**±1900**	**±80**	**0.658**	**85**	**34**
Omega85 IP65/IP68	**±1900**	**±80**	**1.91**	**93**	**39**
Theta	**±2500**	**±400**	**4.99**	**150**	**61**
Theta IP60	**±2500**	**±400**	**8.62**	**190**	**74**
Theta IP65/IP68	**±2500**	**±400**	**9**	**160**	**75**
Omega160	**±2500**	**±400**	**2.72**	**160**	**56**
Omega160 IP60	**±2500**	**±400**	**7.67**	**190**	**58**
Omega160 IP65/IP68	**±2500**	**±400**	**7.26**	**170**	**66**
Omega190	**±7200**	**±1400**	**6.35**	**190**	**56**
Omega190 IP60	**±7200**	**±1400**	**14.1**	**240**	**74**
Omega190 IP65/IP68	**±7200**	**±1400**	**13.2**	**200**	**75**
Omega250 IP60/IP65/IP68	**±16000**	**±2000**	**31.8**	**260**	**95**
Omega331	**±40000**	**±6000**	**47**	**330**	**110**

Tabla 2.3. Modelos de "ASI Industrial automation" y "Schunk" ([5], [6]).

No tenemos ningún sensor que nos proporcionan un rango superior a ± 325 N e inferiores a ±500 para eje "x" e "y", el único que se puede considerar es el Mini 45 (Tabla 2.4), que nos proporciona un máximo valor de rango de 580 N, lo que no supone un excesivo sobredimensionamiento sobre las especificaciones preestablecidos.

Mini 45 de ASI Industrial Automation

Dimensiones

Peso	0.0917 kg
Diámetro	45 mm
Altura	16 mm

- **Material:** Acero inoxidable endurecido
- Alta elación Señal/ruido(SNR), es decir, poca influencia de ruido
- **Aplicaciones:**
  - Telerobóticas
  - Cirugía
  - Mano robótica

	Rango	**Sobrecarga máxima**
Fx, Fy	±580 N	±5100 N
Fz	±1160 N	±10000 N
Tx,Ty	±20 Nm	±110 Nm
Tz	±20Nm	±110 Nm

Tabla 2.4. Características Mini 45 de ASI Industrial Automation.

Una hipotética selección de entre estas dos empresas radicaría en las condiciones de compra que nos ofertase cada una: precio, garantía, mantenimiento, accesorios.

En detrimento de estas compañías nosotros hemos elegido a la JR3 inc. como la suministradora del sensor de fuerza/par que vamos a utilizar, siendo el modelo elegido

el 85M35A3-I40- DH12 para los ensayos en laboratorio. La empresa JR3 inc. fabrica sensores de 6 GDL y de 12 GDL, ambos miden simultáneamente la fuerza y el momento a lo largo y alrededor de los tres ejes ortogonales, los de 12GDL además miden las aceleraciones lineales y angulares en los tres ejes xyz. Nuestro sensor tiene 12 GDL pero únicamente diseñaremos la adquisición de fuerza y pares que es lo que nos interesa en nuestro estudio.

Los sensores JR3 se basan en una célula de carga triaxial y se fabrican en aluminio, en acero inoxidable ó en titanio. En la mayoría de los modelos los datos analógicos se convierten a formato digital mediante sistemas electrónicos incluidos en el sensor. Opcionalmente algunos modelos proporcionan la salida analógica en lugar de la digital [7].

Como los dos anteriores fabricantes de sensores, JR3 inc. establece a XY como el plano horizontal medio del cuerpo del sensor, y al eje "z" como su eje central (Anexo 5). El origen de este sistema de coordenadas es el punto de referencia que el sensor toma para todas las mediciones [7].

JR3 proporciona dos líneas de modelos: La familia "M" y la familia "E".

- **Serie M**: El número anterior a la letra M define el diámetro nominal en mm, y el posterior el espesor nominal del sensor igualmente en mm. Por ejemplo, nuestro sensor seleccionado, 85M35A3-I40- DH12, tiene un diámetro nominal de 85 mm y un espesor nominal de 35 mm.

  Los sensores M incluyen electrónica interna para una mayor inmunidad al ruido, opción de salida digital para usar con una tarjeta de adquisición de datos PCI de JR3 inc., opción de salida analógica, y una configuración electrónica de medio puente. Sus especificaciones típicas son:

  - **Exactitud nominal**: 1% del FS.
  - **Linealidad**: 0,5% FS dentro del rango y del 0,1% por debajo de 1/4 de FS.

- **Resolución**: 1/4000 FS.
- Repetibilidad mejor que precisión absoluta.

- **Serie E**: La serie "E" son sensores de alta precisión diseñados para las aplicaciones más exigentes.

  En la designación de cada sensor el número anterior a la letra "E" indica el diámetro nominal en pulgadas y el número posterior el espesor en pulgadas. Su característica principal diferenciadora es que poseen un puente completo en su configuración electrónica, frente al medio puente de la serie M, que le proporciona la alta precisión que les caracteriza.

  Al igual que la serie M, los sensores E poseen salida analógica y digital, Las especificaciones típicas son:

  - **Precisión nominal**: 0,25% FS.
  - **Linealidad:** 0,5% del FS dentro del rango y del 0,1% de FS por debajo de 1/4 FS.
  - **Resolución**: 1 / 8000 FS
  - Repetibilidad mejor que la precisión absoluta.

Como se observa, la única diferencia entre la serie "M" y la "E" es la mayor precisión de los sensores de tipo "E". Debido a ello, los sensores de la serie "E" son más caros que los "M" (el coste de los modelos E varía de los 5.285 $ a los 5915 $, frente los 4.485 a 4.755 $ de los "M" [8]). Como la precisión que nos ofrece la serie M es perfectamente válida para nuestros necesidades, hemos seleccionado un sensor de esta familia debido a su menor coste. En la Tabla 2.5 [8] se muestra el rango máximo en Fx para la familia "M" de JR3 inc.

Modelo	Valor máximo Fx
50M31	±200 N ±580 N en acero inoxidable
67M25	±400 N ±1200 N en acero inoxidable
90M31	±400 N ±1200 N en acero inoxidable
90M40	±800 N ±2000 N en acero inoxidable
100M40	±800 N ±2000 N en acero inoxidable
160M50	±2000 N ±6300 N en acero inoxidable

Tabla 2.5. Rango máximo de la fuerza en "z" para sensores Serie "M" de JR3.

Siguiendo los requerimientos ya descritos con anterioridad, buscamos un sensor sea capaz de medir en el eje "z" fuerzas superiores a 650 N, imponemos un margen de error y seleccionaremos aquéllos con un máximo valor de rango en torno a los 1000N. De esta forma, atendiendo a la Tabla 2.5, cumplen el requisito los sensores 67M25, 90M31, 90M40, 100M40, 160M50. De ellos, seleccionamos los dos primeros (67M25 y 90M31) porque con los otros sobredimensionaríamos en demasía el diseño. Tenemos que ambos cuestan lo mismo así que la elección de uno u otro se basará en sus dimensiones.

Para nuestros ensayos en el laboratorio no necesitamos un sensor que nos

proporcione tanto rango de medida. Trabajaremos con el modelo 85M35A3-I40- DH12 de JR3 inc. cuyos fondos de escalas según eje se encuentran en la Tabla 2.6:

	Fondo de escala
Fx	63.0 N
Fy	63.0 N
Fz	126.0 N
Mx	4.4 Nm
My	4.4 Nm
Mz	4.4 Nm

Tabla 2.6. Fondos de escala del sensor fuerza/par 85M35A3-I40- DH12 de JR3 inc.

2.4. Teoría del equilibrio bípedo

Caminar es mucho más complejo de lo que parece, ya que no sólo se trata de dar zancadas, si no que este proceso requiere de una coordinación adecuada de pies, cadera, piernas, brazos, hombros, cabeza.

Como el objetivo global de la creación del RH-2 es conseguir que éste camine, a continuación explicamos ante que situaciones a solventar se va encontrar el robot durante una complicada caminata humana.

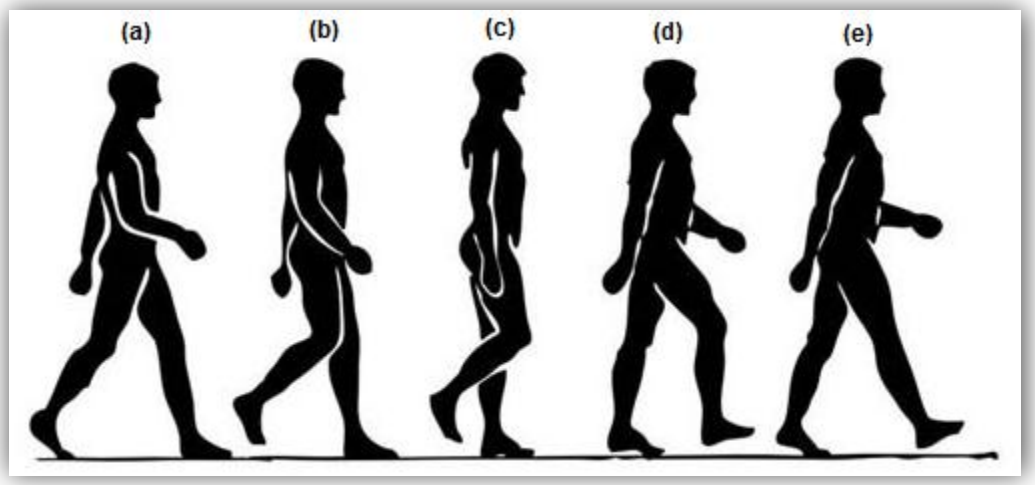


Figura 2.11. Medio ciclo de caminata humana.

La caminata comienza con los dos pies extendidos sobre el suelo (Figura 2.11 a), situación que no presenta inconveniente alguno porque es una configuración anatómica que proporciona un equilibrio estable. El problema comienza en la fase de balanceo cuando uno de los pie se eleva del suelo (Figura 2.11 b, c, d), ya que la tendencia es a caer pues el robot se sustenta únicamente sobre un apoyo. Para evitar que el humanoide

caiga se deben corregir movimientos para permitir la estabilidad dinámica del robot durante la caminata. El medio ciclo se completa cuando los dos pies vuelven a estar ambos apoyados en el suelo (Figura 2.11 e). El otro medio ciclo es igual pero ahora el pie de apoyo es el contrario.

Hay que destacar que durante todo este proceso, el centro de gravedad, así como las diferentes velocidades y aceleraciones de las múltiples articulaciones, varían continuamente. Por ejemplo, en la Figura 2.12 vemos que la fase de balanceo se subdivide en tres fases: en el primer tercio del balanceo, denominado periodo de aceleración, la pierna que abandona el suelo sufre una fuerte aceleración inmediatamente después de que los dedos dejan el suelo. Tras el adelantamiento a la pierna en apoyo, entra en una sub-fase final protagonizada por una desaceleración.

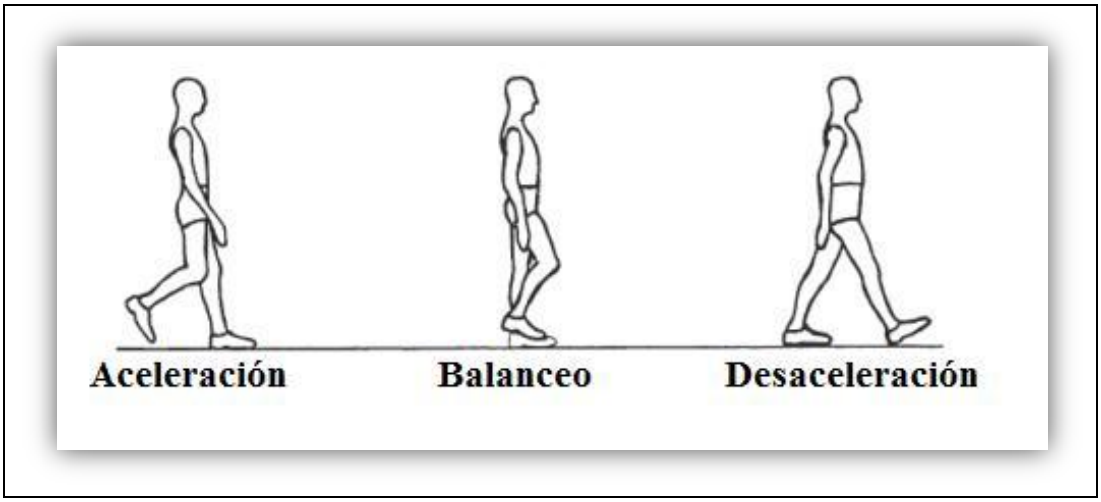


Figura 2.12. Fase de balanceo en caminata humana.

Por su parte, los pies también complican, si cabe, aún más todo este mecanismo de caminata, pues la fuerza de reacción en la planta del pie correspondiente varía notablemente dependiendo de la fase de desplazamiento en que se encuentre. Así, tenemos, como se observa en la figura 2.13, que el movimiento del pie durante el paso se puede subdividir en cuatro intervalos, cada uno con sus singularidades.

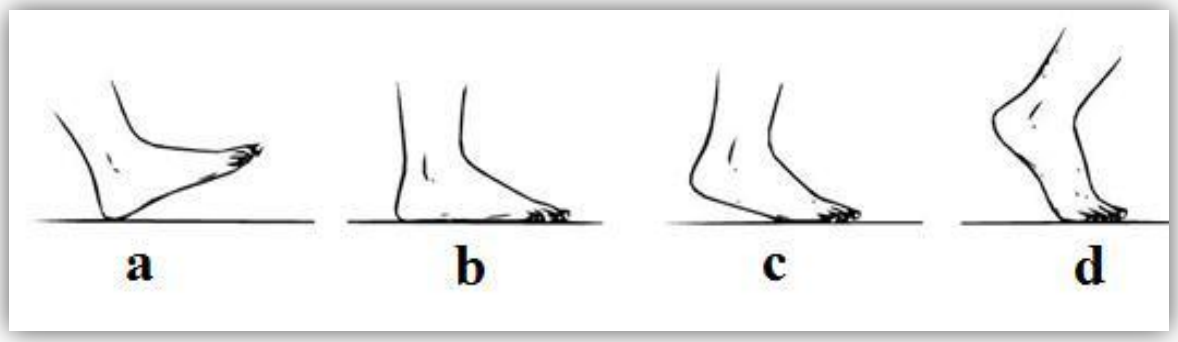


Figura 2.13. Fase de apoyo del pie durante caminata.

- Figura 2.13 (a). Intervalo de aceptación del peso, que empieza con contacto del talón y termina con el apoyo plantar.
- Figura 2.13 (b). El pie permanece plano.
- Figura 2.13 (c). Despegue del talón. Todo el peso reposa sólo sobre el antepié.
- Figura 2.13 (d). Despegue de los dedos.

Este sería el esquema del desarrollo del paso para un pie estrictamente normal. En caso que se amputaran todos los dedos, la última fase del paso no se podría realizar, y el paso se acortaría.

A todos estos movimientos del tronco inferior hay que unir los de la parte superior. Estos últimos, al compensar los movimientos del tren inferior, son esenciales para mantener el cuerpo en equilibrio de rotación. Y es que el movimiento de recuperación de una pierna se compensa con el movimiento hacia abajo del brazo opuesto, mientras que los movimientos de apoyo e impulso son equilibrados alzando el brazo opuesto. Los hombros y torso también se ven involucrados. Estos movimientos del tronco superior serán más exagerados cuanto menos eficientes sean los movimientos de las articulaciones inferiores. Teniendo en cuenta esto último y que la mayor parte de

la energía gastada en la caminata es para equilibrar los movimientos, se buscará eliminar aquellos de la parte inferior del humanoide que sean incorrectos y, en consecuencia, derrochadores [9].

En definitiva, como se puede entender fácilmente ante lo visto, la caminata humana es un tema muy complicado de estudio, de ahí, que se intente simplificar al máximo posible. En este sentido y volviendo a la explicación del ciclo de caminata humana, podemos observar que en la mayor parte de este movimiento el robot se encuentra sustentado por único pie, apoyo sobre el que se balancea de adelante-atrás y/o de izquierda/ derecha, asemejándose a un péndulo invertido. Así pues, el movimiento de caminar de un robot se puede simplificar con el estudio de un péndulo invertido. Basados en el movimiento del péndulo invertido existen, como se muestra en la Figura 2.14, tres modelos para el control de la caminata de un humanoide.

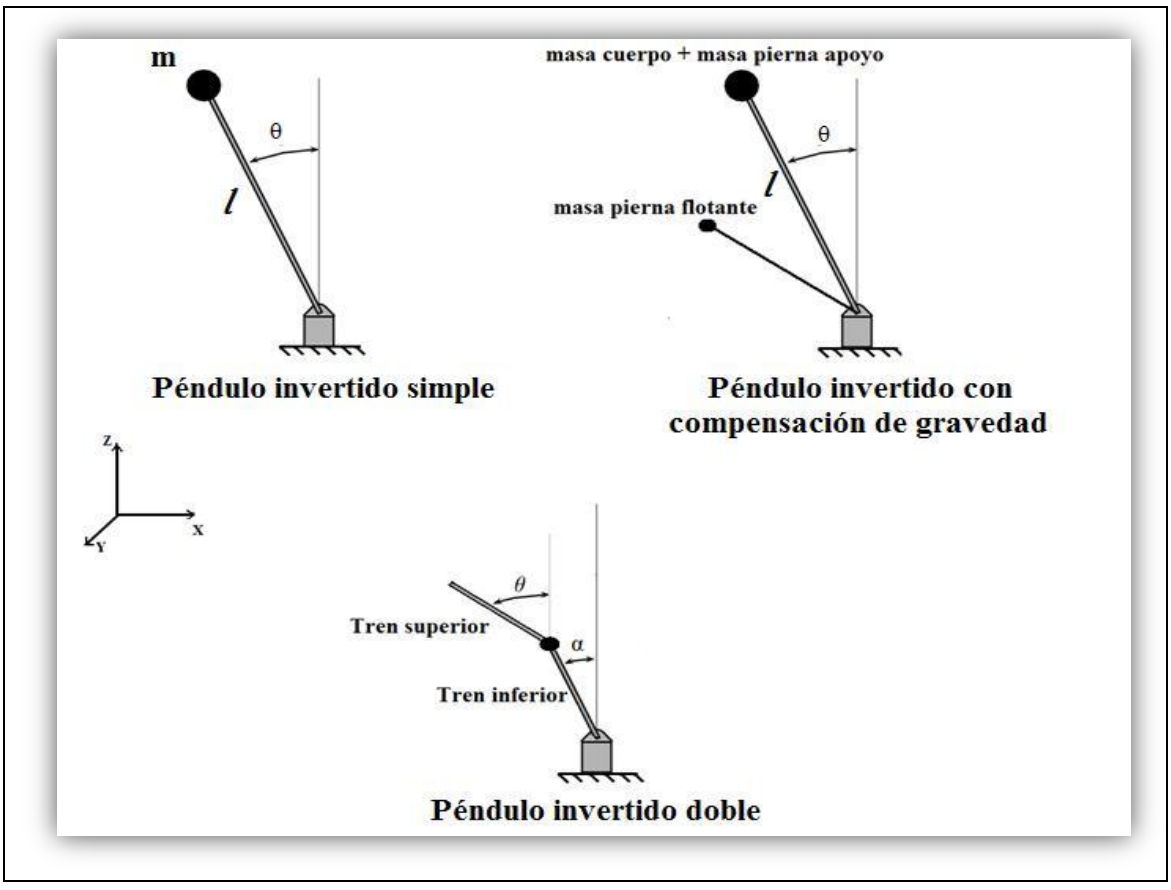


Figura 2.14. Modelos simplificados para la caminata bípeda.

El modelo más sencillo es el del péndulo simple que considera la masa del robot concentrada en su centro de masas, pero presenta la desventaja de que no tiene en cuenta la acción de la pierna flotante que modofica significativamente la dinámica del robot. El segundo modelo sí que contempla la acción de la pierna flotante. Por su parte, el modelo de péndulo doble separa en un segundo eslabón diferente la acción del tronco superior.

Nostros elegimos el modelo más sencillo, es decir, el del péndulo simple. La semejanza entre este sistema y el robot se basa en las siguientes suposiciones:

- La masa del humanoide *m* se concentra en su centro de masas (extremo superior de la barra) a una distancia *l* del suelo.
- La masa de la barra rígida de longitud *l* es despreciable respecto a la masa puntual *m*.
- La acción (par *T*) que permite moverse a la masa *m* un determinado ángulo θ a una cierta velocidad $\dot{\theta}$ (movimiento del centro de masas durante la caminata) se ejerce mediante un servomotor anclado al extremo fijo de la barra. Ese servomotor emula el comportamiento del tobillo del humanoide, que es quien ejerce mayoritariamente la acción de control durante la caminata.

En conclusión, bajo los anteriores supuestos podemos encontrar un modelo de péndulo invertido simple que represente de forma válida a nuestro humanoide RH-2. A continuación se describe dicho modelo. Para escribir la ecuación de movimiento del péndulo [10] se estudian primeramente las fuerzas que actúan sobre el mismo. Existe una fuerza gravitacional igual a *mg*, donde *g* es la aceleración de la gravedad. Igualmente existe otra fuerza de rozamiento contraria al movimiento, que asumimos proporcional a la velocidad del péndulo con un coeficiente de rozamiento *k*. Usando la segunda ley de movimiento de Newton, la ecuación de movimiento en la dirección tangencial puede escribirse como [11]:

$$ml\ddot{\theta} = -mg\sin\theta - kl\ddot{\theta} \qquad \text{(ec. 2.1)}$$

Para la obtención del modelo en espacio de estado se toman como variables $x_1 = \theta$ y $x_2 = \dot{\theta}$. Por tanto, las ecuaciones de estado,

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -\frac{g}{l}\sin x_1 - \frac{k}{m}x_2 \qquad \text{(ec. 2.2)}$$

Para los puntos de equilibrio se cumple $x_1 = x_2 = 0$, resolviendo para $x_1 \; y \; x_2$

$$0 = x_2$$

$$0 = -\frac{g}{l}\sin x_1 - \frac{k}{m}x_2 \qquad \text{(ec. 2.3)}$$

Los puntos de equilibrio se localizan en (n, π), para n = 0, ±1, ±2,...

De la descripción física del péndulo es claro que solo pueden darse dos posiciones de equilibrio que se corresponden con los puntos de equilibrio (0,0) y (π ,0). El resto de puntos de equilibrio son repeticiones de estos dos que se corresponden con el número de vueltas completas que el péndulo barre antes de descansar en alguna de las dos posiciones de equilibrio. Puede observarse que ambas posiciones de equilibrio son muy diferentes, mientras que el péndulo puede reposar en la posición (0,0), éste difícilmente puede mantenerse en la posición (π, 0) ya que una perturbación infinitesimal en torno a ese punto lo desequilibrará. Por tanto, la diferencia entre ambos puntos radica en su estabilidad [11].

Si tenemos en cuenta el par *T* aplicado sobre el péndulo, que en nuestro caso es considerado como la entrada de control, las ecuaciones de movimiento quedan ahora como [11]:

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -\frac{g}{l}\sin x_1 - \frac{k}{m}x_2 + \frac{1}{ml^2}T \qquad \text{(ec. 2.3)}$$

El modelo que se acaba de exponer no es lo sufiente complejo para simular la dinámica completa del humanoide al no considerar la acción de la pierna flotante. Por ello, con el fin de solucionar los problemas derivados de esta aproximación, se propone desarrollar en otros trabajos, el calculo dinámico durante la caminata del COM (centro de masas) a partir de la posición de cada una de las articulaciones [4], como sigue:

$$\overline{x} = \frac{\sum m_i \cdot x_i}{\sum m_i}; \quad \overline{y} = \frac{\sum m_i \cdot y_i}{\sum m_i}; \quad \overline{z} = \frac{\sum m_i \cdot z_i}{\sum m_i} \qquad \text{(ec 2.4)}$$

Siendo $\overline{x}$, $\overline{y}$, $\overline{z}$ las coordenas del COM del robot durante su movimento, mientras que (x_i, y_i, z_i) las coordenas de los centros de masas de cada uno de los eslabones. El origen de coordenadas se toma en el pie de apoyo.

La inclinación absoluta del cuerpo se puede calcular mediante un sensor inercial (SMI), ha caracterizar en otros trabajos.

2.5. Criterio estabilidad ZMP(Punto de momento cero).

La idea fundamental del control de equilibrio, como es obvia, es evitar que el humanoide caiga. Para ello se han desarrollado distintos indicadores, de entre todos ellos, nosotros nos vamos a ocupar del ZMP (zero momento point-punto de momento cero).

El concepto ZMP fue introducido en 1968 por el serbio Miomir Vukobratović en el Tercer Congreso de Mecánica Teórica y Aplicada [12]. Su utilización como técnica de control de equilibrio fue aplicada con éxito por primera vez en 1984 en Waseda University, Laboratory of Ichiro Kato, en el robot WL-10RD, el primero equilibrado dinámicamente, y desde entonces en otros múltiples robots. Dicha técnica de control basada en el cálculo del ZMP establece un criterio de estabilidad dinámica para el robot que permite generar patrones de locomoción.

EL ZMP se puede definir como el punto dentro de la superficie de contacto en el que las fuerza externas no producen momento neto en la dirección horizontal. El concepto supone que el área de contacto es plana y tiene una fricción suficientemente alta como para impedir que los pies deslicen. Cuando el ZMP se encuentra dentro del polígono de soporte, el contacto entre el suelo y el pie es estable. El polígono de soporte es el área conexa formada con los puntos de contacto sobre el suelo (ver Figura 2.15). En caso de soporte sobre un único pie, el polígono de soporte es este mismo. En caso de soporte con los dos pies, el polígono de soporte abarca la región de los pies y la parte del área entre ambos. Cuanto más cercano esté ZMP al centro del polígono de soporte, mayor equilibrio se consigue. Cuando el punto de momento nulo está fuera de esta superficie de soporte, el robot se inclina rotando sobre alguno de los bordes de dicho polígono. El criterio de que ZMP exista dentro del polígono de soporte es condición necesaria y suficiente para garantizar la estabilidad dinámica del robot.

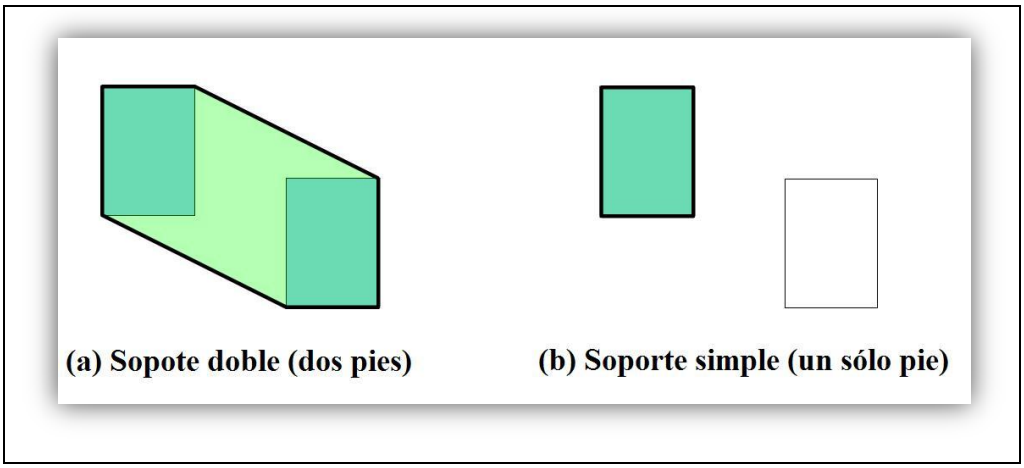


Figura 2.15. Polígono de soporte según fase de caminata.

2.5.1. Cálculo matemático del ZMP

A continuación se hace una demostración sencilla del cálculo del punto ZMP:

Partamos del análisis de todas las fuerzas y momentos que actúan sobre el pie de apoyo, considerando éste como un sólido rígido (Figura 2.16).

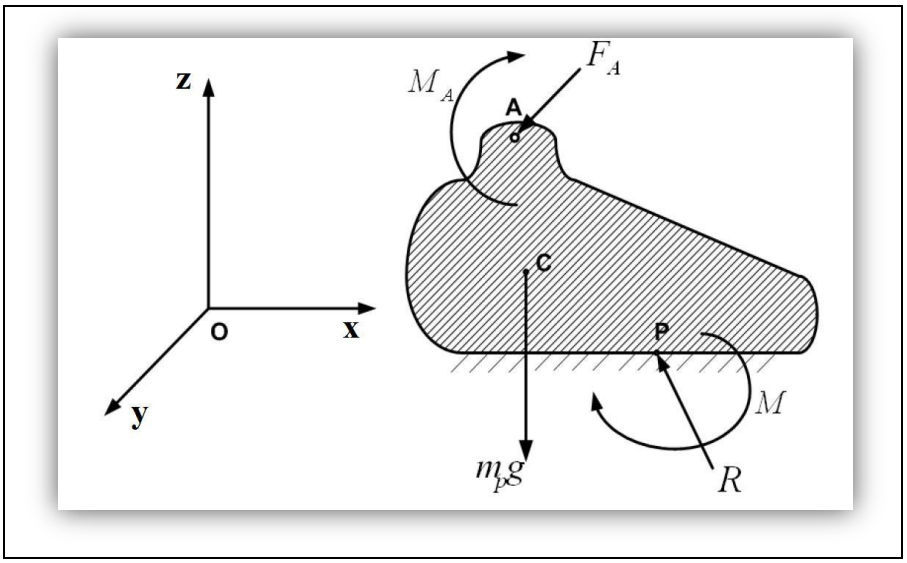


Figura 2.16. Fuerzas y momentos sobre el pie de apoyo.

De la Figura 2.17 se tiene que F_A y M_A son la fuerza y el momento resultante generado por el cuerpo sobre su tobillo, mientras que R y M son la fuerza de reacción y momento del suelo. Por su parte, m_p es la masa del pie.

Para que el pie esté en equilibrio estático se debe cumplir:

- $\sum \vec{F} = 0 \Rightarrow \overrightarrow{R} + \overrightarrow{F_A} = 0$
- $\sum M_O = 0 \Rightarrow \overrightarrow{OP} \times \overrightarrow{R} + \overrightarrow{OA} \times \overrightarrow{F_A} + \overrightarrow{OC} \times (m_p \cdot \vec{g}) + \vec{M} + \overrightarrow{M_A} = 0$

(ec. 2.5)

Supongamos ahora que la fricción de la superficie es lo suficientemente grande como para no permitir el deslizamiento entre el pie y el suelo. Esto se traduce en que,

- F_{Ax} y F_{Ay} se ven compensados por la reacción del suelo R_x y R_y,
- Igualmente, M_{Az} se compensa por el momento en el eje Z producido por el suelo (M_z)

Esto se expresa en las dos siguientes igualdades:

$$\boldsymbol{R_x = -F_{Ax}} \quad ; \quad \boldsymbol{R_y = -F_{Ay}}$$
$$\boldsymbol{M_z = -M_{Az}} \qquad \text{(ec. 2.6)}$$

De esta forma, la ecuación 1 es reducida a la componente vertical de las fuerzas, mientras que la ecuación 2 se simplifica a la componente horizontal (H) de los momentos, teniendo:

- $\sum F_z = 0 \Rightarrow R_z + F_{Az} = 0 \Rightarrow R_z = -F_{Az}$ (ec. 2.7)
- $(\sum M_O)^H = 0 \Rightarrow$

$$\Rightarrow \left(\overrightarrow{OP} \wedge \vec{R}\right)^H + \left(\overrightarrow{OA} \wedge \overrightarrow{F_A}\right)^H + \left(\overrightarrow{OC} \wedge (m_p \cdot \vec{g})\right)^H + \left(\vec{M}\right)^H + \left(\overrightarrow{M_A}\right)^H = 0 \qquad \text{(ec.2.8)}$$

De las ecuaciones 2.6 y 2.7, tenemos que la reacción del suelo es:

$$\vec{\boldsymbol{R}} = -\left(\boldsymbol{F_{Ax}}\vec{\boldsymbol{\imath}} + \boldsymbol{F_{Ay}}\vec{\boldsymbol{\jmath}} + \boldsymbol{F_{Az}}\vec{\boldsymbol{k}}\right) \qquad \text{(ec. 2.9)}$$

La condición fundamental para que el sistema esté en equilibrio es que el pie en el punto P (Figura 2.17) no rote, es decir,

$$\boldsymbol{M_x} = \boldsymbol{M_y} = \boldsymbol{0} \Rightarrow \boldsymbol{M^H} = \boldsymbol{0} \qquad \text{(ec 2.10)}$$

Teniendo en cuenta la ecuación 2.10, cambiando el punto de referencia O por A (Figura 2.17), y despreciando la masa del pie, la ecuación 2.8 queda:

$$\left(\sum M_A\right)^H = 0 \Rightarrow$$

$$\Rightarrow \left(\overrightarrow{AP} \times \vec{R}\right)^H + \left(\overrightarrow{AA} \times \overrightarrow{F_A}\right)^H + \left(\overrightarrow{M_A}\right)^H = \left(\overrightarrow{AP} \times \vec{R}\right)^H + \left(\overrightarrow{M_A}\right)^H = 0 \Rightarrow$$

$$\left(\overrightarrow{\boldsymbol{AP}} \times \vec{\boldsymbol{R}}\right)^{\boldsymbol{H}} = -\left(\overrightarrow{\boldsymbol{M_A}}\right)^{\boldsymbol{H}} \qquad \text{(ec. 2.11)}$$

Claramente la ecuación 2.11 se cumple eligiendo el punto P adecuado, de tal forma, que el momento generado por la fuerza de reacción del suelo, aplicada en P, compense al momento total generado por el cuerpo en su caminata. Este punto P es el denominado punto de momento nulo, el ZMP.

De la ecuación 2.11 obtenemos la expresión que define al ZMP,

$$\left(\overrightarrow{AP} \times \vec{\boldsymbol{R}}\right)^H = -\left(\overrightarrow{M_A}\right)^H \Rightarrow \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ p_x & p_y & p_z \\ R_x & R_y & R_z \end{vmatrix} =$$

$$= \vec{\imath} \cdot \left(p_y \cdot R_z - p_z \cdot R_y\right) - \vec{\jmath} \cdot \left(p_x \cdot R_z - p_z \cdot R_x\right) = -\left(\overrightarrow{M_A}\right)^H \Rightarrow$$

$$\Rightarrow \begin{cases} -(p_x \cdot R_z - p_z \cdot R_x) = -M_{Ay} \Rightarrow p_x = \dfrac{M_{Ay} + p_z \cdot R_x}{R_z} \\ p_y \cdot R_z - p_z \cdot R_y = -M_{Ax} \Rightarrow p_y = \dfrac{-M_{Ax} + p_z \cdot R_y}{R_z} \end{cases} \quad \text{(ec 2.12)}$$

Sustituyendo la ecuación 2.9 en 2.12, tenemos finalmente la expresión que define al ZMP:

$$\boldsymbol{ZMP = (ZMP_x, ZMP_Y) = \left(\frac{p_z \cdot F_{Ax} - M_{Ay}}{F_{Az}}, \frac{p_z \cdot F_{Ay} + M_{Ax}}{F_{Az}}\right)} \qquad \textbf{(ec.2.13)}$$

3. Elementos de diseño

Trabajamos sobre un péndulo simple invertido en cuya base se encuentra nuestro sensor de fuerza/par de la compañía JR3 a caracterizar. Mediante un cable modular RJ-11 de 6 pines el sensor está conectado a la tarjeta de adquisición de datos PCI P/N 1593 de JR3 inc. instalada en un PC con sistema operativo Linux (kernel 2-6.26-2-686). Todo este conjunto se muestras en la Figura 3.1

Figura 3.1. Elementos de diseño. En sentido de las agujas del reloj: Conjunto péndulo con ordenador con Linux (kernel 2-6.26-2-686); sensor fuerza/par 85M35A3-I40-DH12 de JR3 inc.; cable modular RJ-11; tarjeta de adquisición de datos PCI P/N 1593 de JR3 inc.

En los siguientes apartados explicaremos cada uno de los elementos que forman nuestro sistema. El orden será: Péndulo invertido simple, sensor de fuerza/par, tarjeta PCI y cable modular RJ-11.

3.1. Péndulo simple invertido

El péndulo invertido simple utilizado, mostrado en la Figura 3.2, tiene dos grados de libertad de rotación, uno con un recorrido de 180º en el plano sagital (Figura 3.3) y otro de 20º en el plano frontal (Figura 3.4). Gracias a estos dos grados, conseguiremos simular el movimiento del tobillo en la fase de simple apoyo durante la caminata [13].

Figura 3.2. Péndulo invertido.

En cuanto a su estructura, es íntegramente metálica y en su extremo superior, a 1 metro de distancia de la base (distancia media al centro de masas de un cuerpo humano), se encuentra situado un cilindro metálico que permite roscar pesas de disco de diferentes masas para probar el prototipo en diversas condiciones de carga [13].

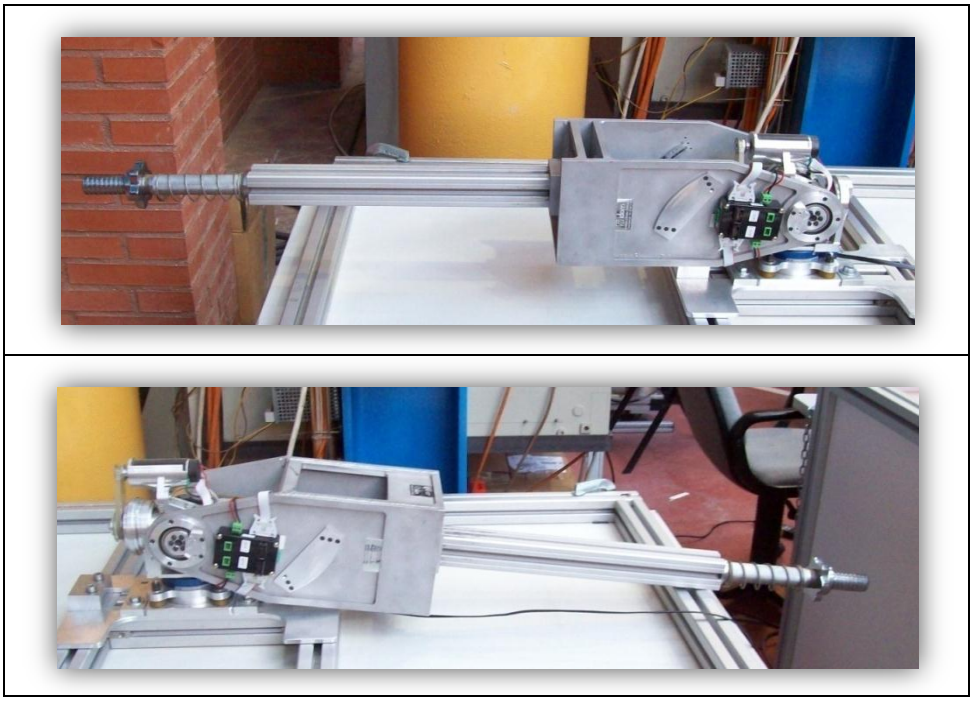


Figura 3.3. Recorrido 180º en el plano sagital del péndulo invertido.

Figura 3.4. Recorrido 20º en el plano frontal del péndulo invertido.

El péndulo dispone de una base para anclaje, con unas dimensiones de 35cm de largo y 21cm de ancho (véase Figura 3.5). Se encuentra situado sobre una mesa, anclado a un banco para evitar los posibles movimientos debidos a las trayectorias que describirá el mismo [13].

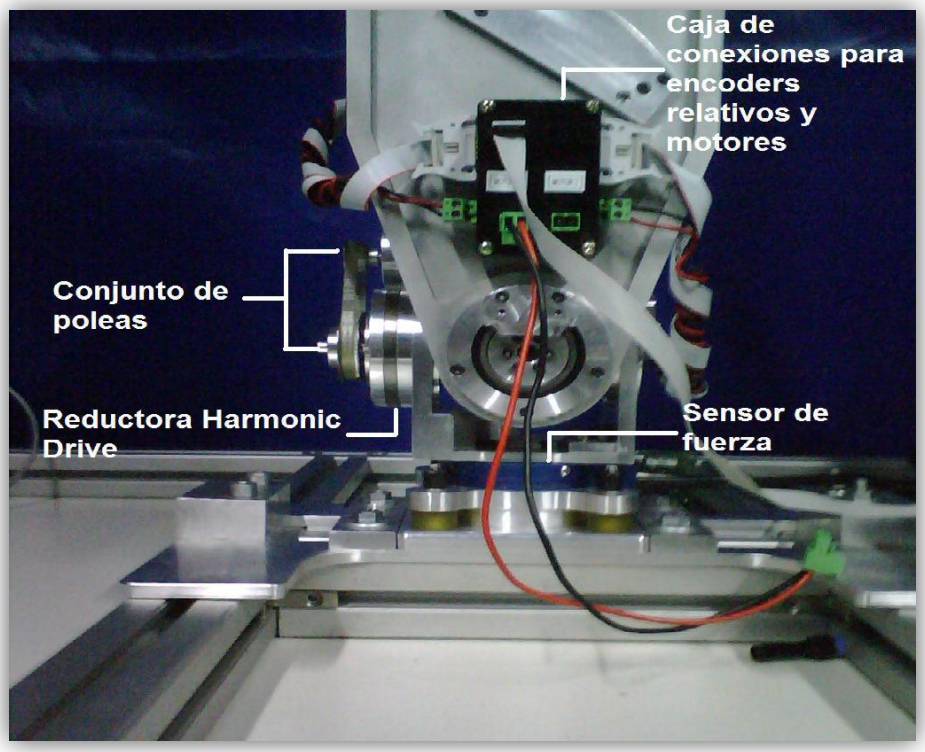


Figura 3.5. Base del péndulo [13].

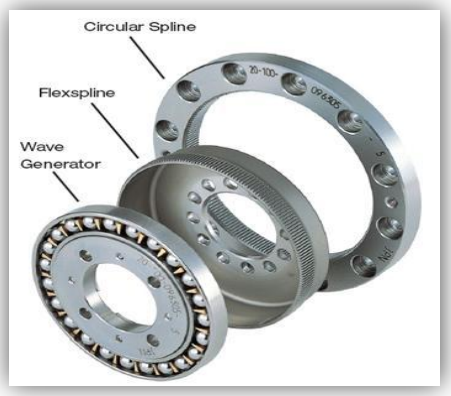
El péndulo se mueve gracias a dos motores de marca Maxon DC (Figura 3.6 a), uno por cada grado de libertad. Para el control de ambos motores se utiliza el driver Technosoft ISCM 8005 (figura 3.6 b). La generación de las trayectorias de los motores se realiza mediante el programa Easy Motion Studio. Los encóders utilizados son del tipo magnéticos resistivos de la casa Maxon (Figura 3.6 c). El equipo de reducción consta de dos reductoras Harmonic Drive modelo CSD-20 con relación de reducción

1/160 (Figura 3.6 d) y cuatro poleas (Figura 3.7 a). La fuente de alimentación utilizada es el modelo SL30 de la casa PULS (figura 3.7 b). Todos estos elementos se conectan a una placa PCB de conexión (figura 3.7 c). Por último, para medir la fuerza y momentos contamos con un juego de sensor y tarjeta PCI a describir en el siguiente apartado.

(a) Motor Maxon DC [13]

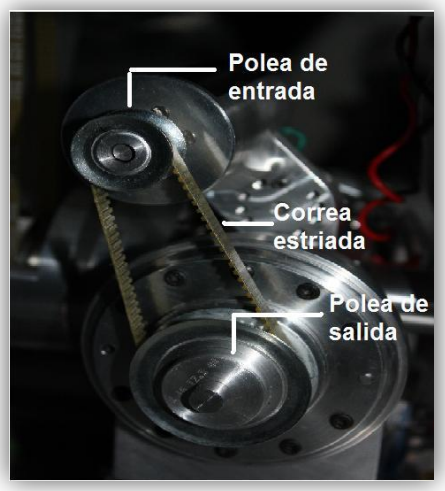
(b) Driver Technosoft ISCM 8005 [13]

(c) Encóder relativo de Maxon [13]



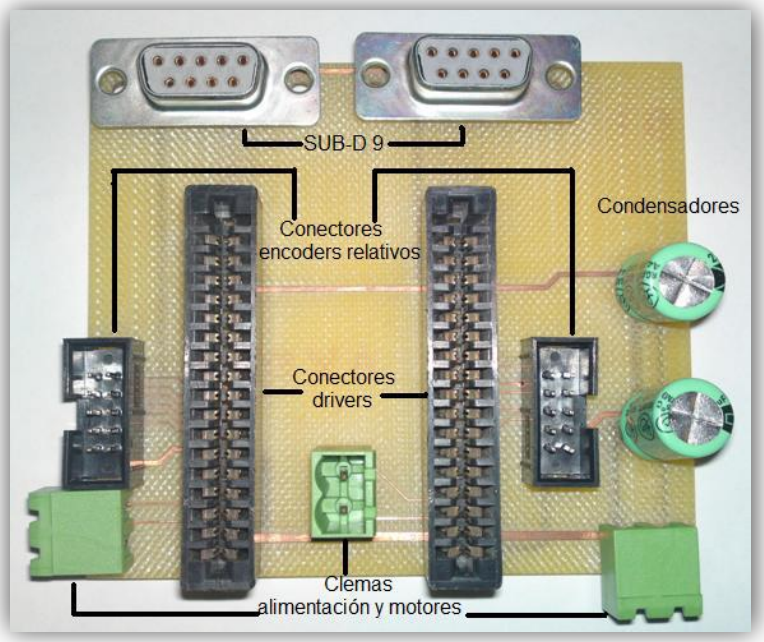
(d) Harmonic Drive modelo CSD-20 [13]

Figura 3.6. Elementos del tobillo: Motores, driver, encóder y Harmonic Drive.



(a) Juego de poleas [13]

(b) Fuente alimentación Modelo SL30 de PULS [13]



(c) Placa PCB de conexión [13]

Figura 3.7. Elementos del tobillo: Juego de poleas, fuente alimentación y placa de conexión

3.2. Sensor fuerza/par 85M35A3-I40-DH12 de JR3 inc.

El péndulo de ensayos utilizado posee en su base un sensor de fuerza-par, cuya finalidad es medir en todo momento las fuerzas y pares que se están realizando sobre la base del péndulo (tobillo del humanoide). En la Figura 3.8 se muestra el péndulo invertido, situando en el mismo la localización del sensor de fuerza/par.

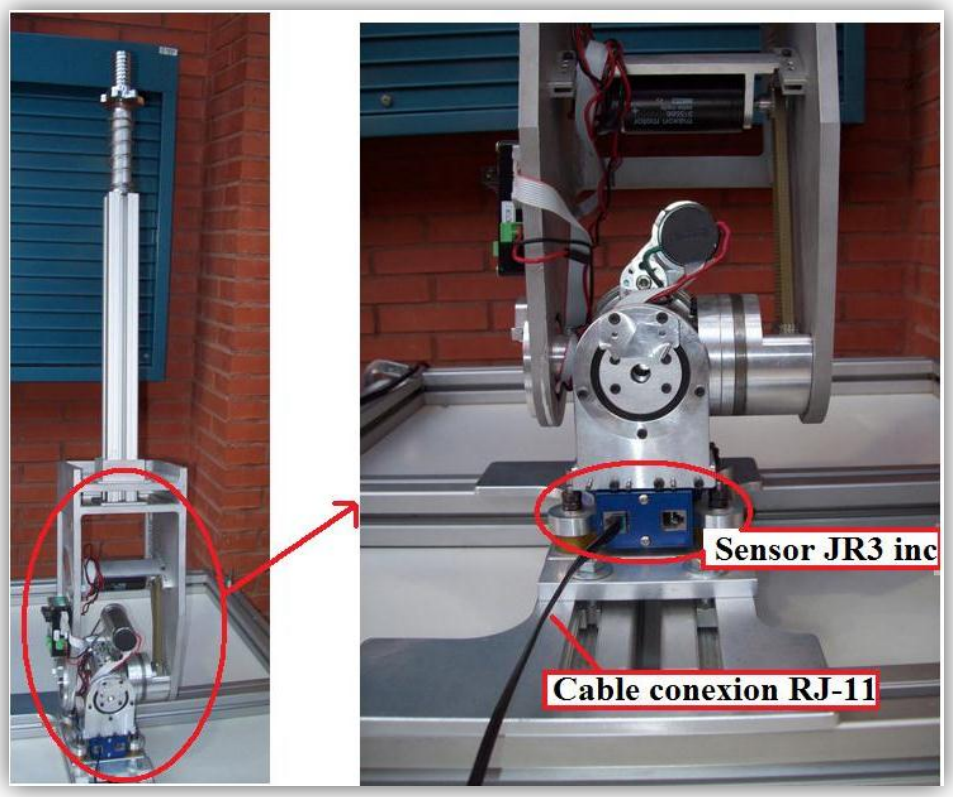


Figura 3.8. Base péndulo invertido donde se ubica el sensor fuerza/par JR3 inc.

Se trata del sensor 85M35A3-I40-DH12 perteneciente a la serie "M" de la compañía JR3 inc. (Figura 3.9), de 12 grados de libertad: Mide la fuerza y el momento a lo largo y alrededor de los tres ejes ortogonales, así como, las aceleraciones lineales y angulares también en los tres ejes. Nosotros únicamente realizaremos una aplicación para medir las fuerzas y momentos.

Figura 3.9. Sensor fuerza/par 85M35A3-I40-DH de JR3 inc.

3.2.1.Características del 85M35A3-I40-DH12 de JR3 inc.

El sensor de fuerza/par 85M35A3-I40-DH12 de JR3 inc. es un dispositivo monolítico de aluminio de 85 mm de diámetro y 35 mm de altura. Contiene sistemas electrónicos analógicos y digitales que están protegidos por el propio cuerpo metálico del sensor de posibles interferencias electromagnéticas. El sistema de medición se basa en galgas de deformación que miden las cargas a las que está sometido el dispositivo JR3. La tensión de estas galgas es amplificada y combinada para crear representaciones analógicas de la carga y los momentos en los tres ejes. Estos datos analógicos son convertidos a la forma digital gracias a los sistemas electrónicos contenidos en el interior del sensor. Como se observa en la Figura 3.9 posee dos salidas, una que proporciona los datos en Sistema Internacional y otra que los proporciona en Sistema Anglosajón.

Sus características son:

- **Exactitud nominal**: 1% del FS.
- **Repetibilidad:** Mejor que precisión absoluta.
- **Linealidad**: del 0,5% FS dentro del rango y del 0,1% por debajo de 1/4 de FS.
- **Resolución**: 1/4000 FS.

En el capítulo 4 estudiaremos experimentalmente su curva de calibración, sensibilidad, linealidad e histéresis.

El sistema de referencia de coordenadas se encuentra centrado en el plano medio del cuerpo del sensor, siendo el XY este plano horizontal, mientras que el eje "z" es el eje vertical del sensor. El punto de referencia para todos los datos de carga es el centro geométrico del sensor correspondiente al origen de coordenadas xyz (ver Figura 3.10 y Anexo 3).

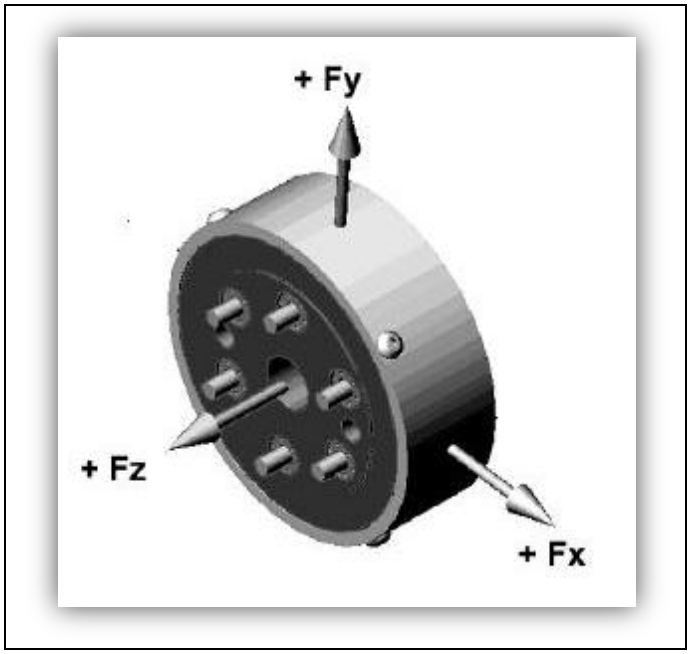


Figura 3.10. Sistema referencia sensor fuerza/par JR3.

3.2.2. Calibración del 85M35A3-I40-DH12 de JR3 inc.

Todos los sensores de varios ejes, como es el caso de nuestro sensor JR3, tienen algún grado de acoplamiento cruzado, es decir, una carga en un eje puede producir un cambio en los valores de los otros ejes. Debido a esto, en el sensor JR3 se debe calibrar cada eje de forma independiente respecto a los otros dos.

3.2.3. Requerimientos de montaje del 85M35A3-I40-DH12 de JR3 inc.

Los sensores JR3 inc. son dispositivos de precisión, por lo que sus medidas pueden falsearse si no tomamos una serie de precauciones durante su montaje. Estas prevenciones consisten en apretar los 4 tornillos del sensor poco a poco en varias etapas

secuenciales, ya que si, de lo contrario, un tornillo se aprieta hasta al fondo dejando los otros flojos, la fuerza ejercida por el tonillo apretado puede ser transmitida al sensor, quedando como consecuencia éste dañado e imposibilitando la medición de la fuerza y/o del par, en por lo menos, uno de los tres ejes. Es decir, para que el sensor mida correctamente se tiene que dar una vuelta a un tonillo, otra vuelta al siguiente y de esta manera hasta completar una vuelta para cada tornillo; en la siguiente etapa, se da una segunda vuelta a cada uno de los 4 tornillos y así sucesivamente hasta que el dispositivo quede bien amarrado al montaje, sin exceder nunca los pares de apriete determinados por el fabricante JR3 inc. para cada tamaño de tornillo. Estos valores máximos se presentan en el Tabla 3.1.

Tamaño cabeza tornillo	M3x.5	M4x.7	M5x.8	M6x1.0	M8x1.25	M10x1.5	M12x1.75	M16x2.0
Par máximo recomendado (Nm)	1.25	2.9	5.9	10	24	48	84	207

Tabla 3.1. Valores de par para el apriete de los tornillos en el montaje de los sensores fuerza/par JR3.

En nuestro caso específico, el tamaño de los tornillos del sensor 85M35A3-I40-DH12 de JR3 inc. es M4, por tanto, viendo la Tabla 3.1, en el montaje del sensor en el péndulo no deberemos sobrepasar los 2.9 Nm en el ajuste de los tornillos.

Otro requerimiento para el buen funcionamiento del sensor es montarlo sobre una superficie extremadamente plana, preferiblemente dentro de .0127 mm. Además, la superficie de montaje debe ser lo suficientemente rígida como para soportar todo el rango de medidas del sensor, para ello estas superficies deben cumplir un mínimo de espesor en función de la máxima carga que mide el dispositivo. Estos espesores, proporcionados por el fabricante, se muestran en la Tabla 3.2.

Fx, Fy (N)	Espesor mínimo (mm)	
	Soporte de aluminio	Soporte de acero
66,75	12,7	9,525
111,25	12,7	9,525
222,5	19,05	12,7
445	22,225	15,875
1112,5	31,75	22,225
2225	38,1	25,4

Tabla 3.2. Recomendación espesor de soporte según carga máxima.

El sensor que utilizamos en nuestro trabajo tiene un rango máximo en "x" e "y", de 63 N, por tanto, nuestra plataforma de agarre deberá tener, según la Tabla 3.2, un espesor mayor a 12,7 mm si se fabricar en aluminio, o de 9,525 mm en caso de fabricarse en acero. Nuestro tobillo está completamente fabricado en aluminio, por tanto, el espesor mínimo de nuestro plato debe ser de 12,7 mm.

Por último, hay que asegurarse de que el espacio de contacto entre el sensor y la superficies de apoyo esté libre de materiales sólidos, ya que de existir impurezas, la carga que producirían éstas durante el correcto apriete de los tonillos podría ser transmitida al sensor dando lugar a posteriores medidas falseadas.

Para la adquisición de los datos medidos por nuestro sensor JR3 utilizaremos una tarjeta PCI PN 1593 de JR3 inc. que a continuación describimos.

3.3. Tarjeta de adquisición de datos PCI PN 1593 de JR3 inc.

En el diseño utilizamos la tarjeta de adquisición de datos PCI PN 1593 de JR3 inc. (Figura 3.11 y Figura 3.12). Dicha tarjeta posee 4 puertos (numerados según Figura 3.11) de tecnología analógica ADSP-2184 de alta velocidad y se conecta con el sensor de fuerza/par mediante un cable modular de 6 u 8 pines (R-11 y R-45), nosotros utilizamos el modelo R-11 (Figura 3.11). La PCI-bus utiliza este cable para recibir a alta velocidad los datos proporcionados por el sensor, así como para suministrar la energía que requiere este sensor para su funcionamiento. La tarjeta consta de un circuito para controlar y ajustar la tensión de alimentación de cada uno de los cuatro posibles sensores conectados. En cuanto a su propia alimentación, la PCI no requiere de fuente externa, ya que la energía que necesita la recibe directamente del ordenador en el que se encuentra instalada. Sus tensiones y corrientes de funcionamiento:

- V - 870 mA típico
- 12 V - 25 mA típico (w / o sensor)
- -12V - 5 mA típico (w / o sensor)

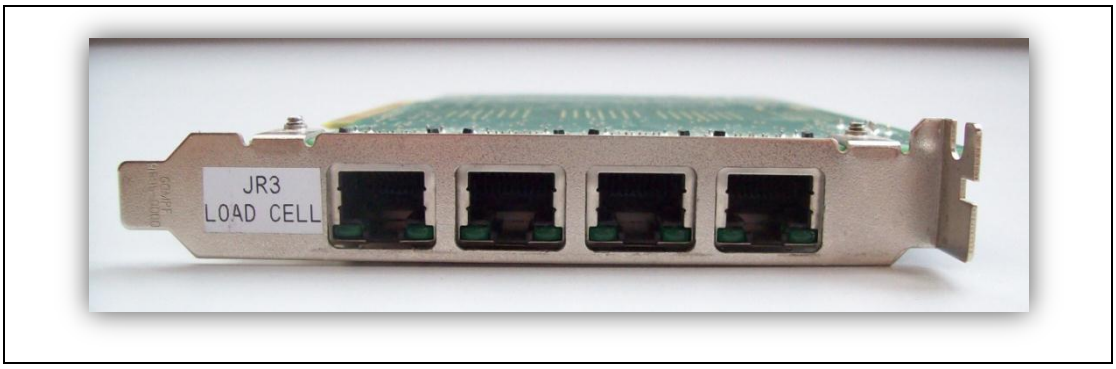


Figura 3.11. Numeración de los puertos. Tarjeta PCI PN 1593 de JR3 inc.

De izquierda a derecha: SENSOR0, SENSOR1, SENSOR2, SENSOR3.

Figura 3.12. Tarjeta de adquisición de datos PCI PN 1593 de JR3 inc.

El acceso a los datos recibidos por la PCI se hace directamente en la memoria de la tarjeta, en la cual cada dato tiene su propia dirección predeterminada. Por ejemplo, alguno de los datos disponibles con sus direcciones correspondientes vienen en la Tabla 3.3:

Addr	Data	Addr	Data
0x60	Shunts	0x68	Default Full Scale
0x70	Min Full Scale	0x78	Max Full Scale
0x80	Full Scale	0x88	Offsets
0x90	Filter 0	0x98	Filter 1
0xa0	Filter 2	0xa8	Filter 3
0xb0	Filter 4	0xb8	Filter 5
0xc0	Filter 6	0xc8	Rates
0xd0	Minimums	0xd8	Maximums

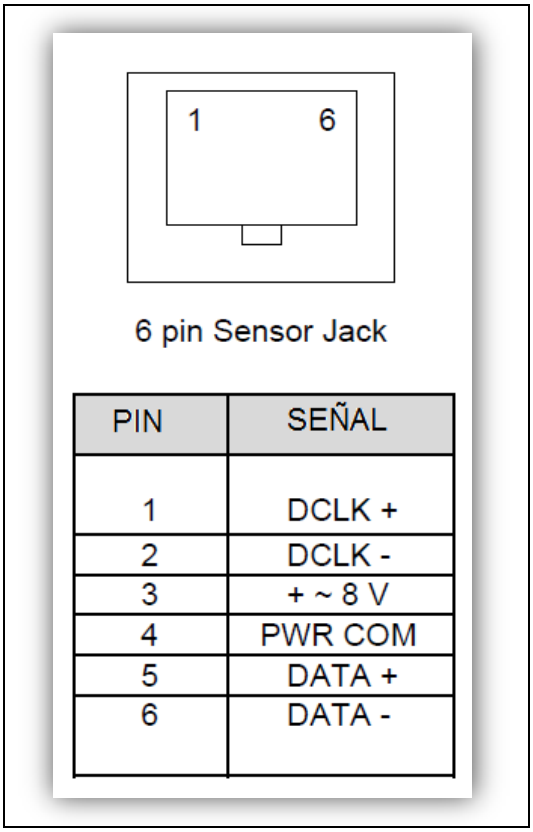
Tabla 3.3. Datos y su dirección de memoria.

La tarjeta adquisición de datos PCI PN 1593 de JR3 inc. la instalamos en un PC (Figura 3.13) con sistema operativo Linux (kernel 2-6.26-2-686).

Figura 3.13. PCI PN 1593 de JR3 inc. instalada en PC.

3.4. Cable modular de conexión RJ-11.

El sensor fuerza/par se conecta a la tarjeta PCI de adquisición de datos a través de un cable modular RJ-11 de 6 pines (ver Figura 3.14). Dicho cable sirve para la transmisión de datos a alta velocidad desde el sensor a la PCI y para que ésta suministre la energía que requiere dicho sensor. Las conexiones del cable RJ-11 están especificadas en la Figura 3.14.



PIN	SEÑAL
1	DCLK +
2	DCLK -
3	+ ~ 8 V
4	PWR COM
5	DATA +
6	DATA -

Figura 3.14. Conexionado RJ-11.

Según las hojas de características adjuntas en el Anexo 3 se debe prestar especial atención a la conexión de estos cables, ya que las señales irán conectadas en pines opuestos en un extremo del cable con respecto al otro, es decir, que la señal que se

conecta en un extremo en el pin 1, en el otro irá al pin 8, el del 2 al 7, y así con todas las conexiones (ver Figura 3.15).

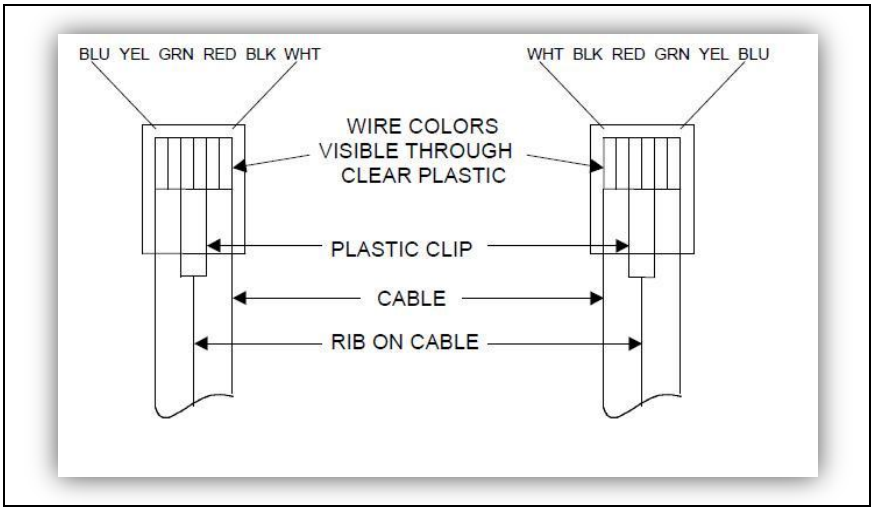


Figura 3.15. Cable modular RJ-11.

4. Desarrollo experimental.

En este capítulo nos centraremos en explicar, en un primer lugar, cómo funcionan las diferentes aplicaciones diseñadas, tanto el código en C++ para la adquisición de datos, así como, el funcionamiento de la interfaz gráfica desarrollada en MATLAB, sin olvidarnos del conjunto de programas (memoria compartida y funciones MEX) utilizados para la comunicación en tiempo real entre la interfaz y el código de adquisición de datos.

En una segunda parte, presentaremos los diferentes ensayos realizados con el fin de verificar que el sensor mide correctamente lo que tiene que medir y caracterizarlo en su medición de fuerzas y momentos en los tres ejes, para lo cual nos ayudaremos del robot ABB que la Universidad Carlos III de Madrid dispone en su Campus de Leganés. Por último, mostraremos los resultados obtenidos de los ensayos llevados a cabo en la plataforma del tobillo.

4.1. Diseño de aplicaciones.

Nuestra aplicación diseñada para la obtención y representación de datos se estructura en tres módulos (Figura 4.1).

1º). Aplicación "**jr3.cpp**" cuya función es la de adquirir los datos de la tarjeta PCI PN 1593 de JR3 inc.

2º). Por otro lado, tenemos la interfaz gráfica de MATLAB "**interfazjr3_display**", que visiona en tiempo real los valores de fuerzas y momentos adquiridos por "jr3.cpp" y que, además, permite el cálculo en tiempo real del ZMP en apoyo simple, la representación a posteriori de la trayectoria del ZMP, del vector fuerza resultante y de las fuerzas y momentos respecto al tiempo.

3º). Como medio de comunicación entre "jr3.cpp" e "interfazjr3_display" está definida una memoria compartida (**"memoria.h"**) a la que tienen acceso los dos programa. Mientras que "jr3.cpp" tiene comunicación, por así definirla, normal a la "memoria.h" por ser código C++, la interfaz requiere de un programa auxiliar (en este caso cuatro, uno por sensor) que le permita tener acceso a los valores guardados en "memoria.h". Así, el grupo de códigos que constituyen el medio de comunicación, lo complementan cuatro funciones MEX (que son funciones escritas en C/C++ y que se ejecutan desde MATLAB) "leer_datso0.cpp", "leer_datos1.cpp", "leer_datos2.cpp" y "leer_datso3.cpp", es fácil entender que cada una de estas funciones tienen el mismo funcionamiento pero para acceder cada una a los datos de su número de sensor correspondiente.

4.1.1. Aplicación "jr3.cpp".

Como se explica en el diagrama de flujo de la Figura 4.2, la función de esta aplicación es leer los valores medidos por el dispositivo sensor; realiza, con los fondos de escala introducidos por el usuario desde la "interfazjr3_display" de Matlab, la transformación de medida eléctricas a unidades internacionales de fuerzas y momentos, para posteriormente guardar estos datos en un archivo, presentarlos por la terminal y guardarlos en la memoria compartida para que la interfaz de Matlab pueda presentarlos en su display. Como se observa, trabaja en un bucle infinito que se inicia cuando se presiona desde la interfaz de Matlab el botón ON y termina cuando se presiona desde la misma interfaz OFF.

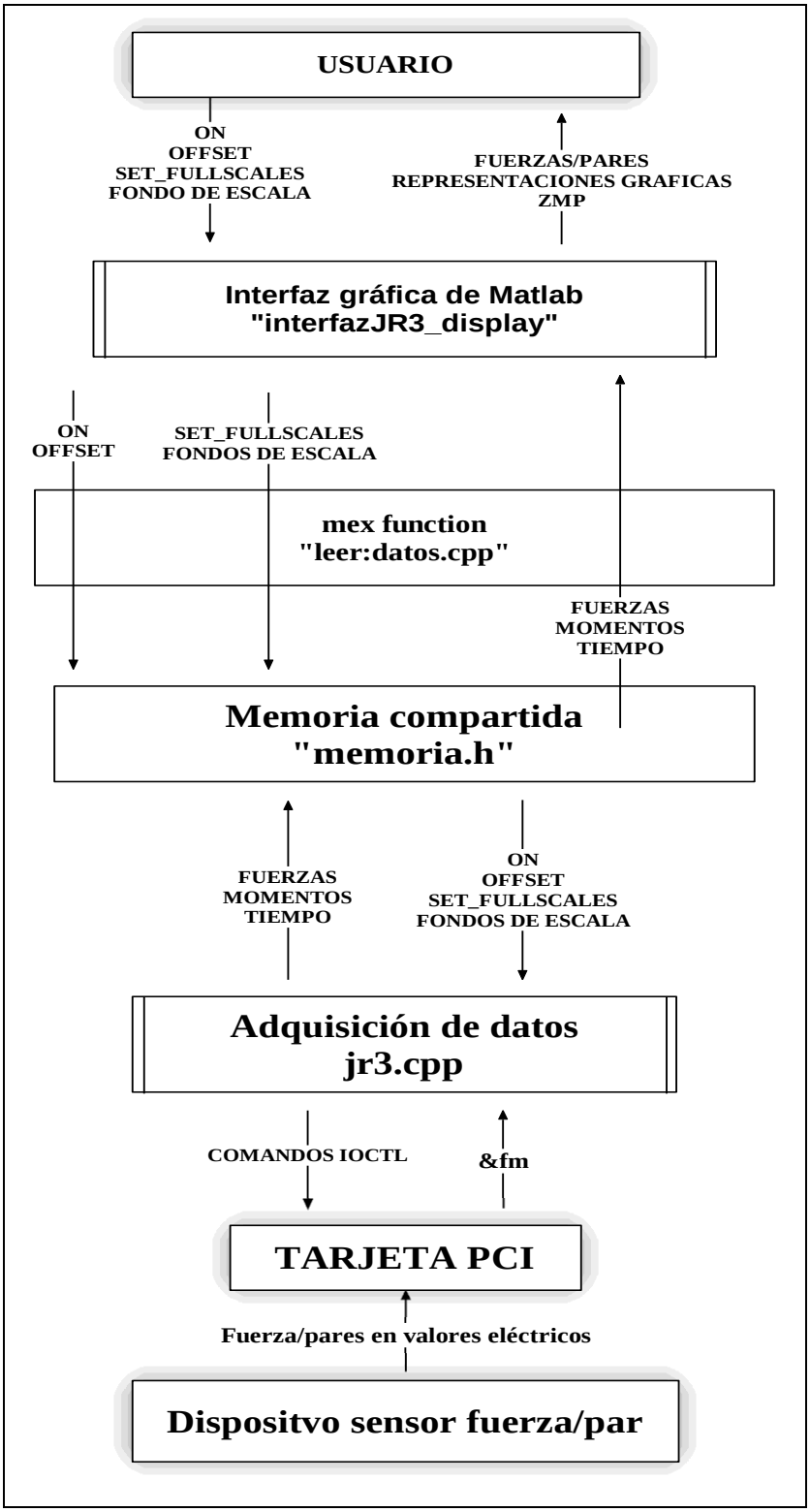


Figura 4.1. Diagrama explicativo aplicación diseñada.

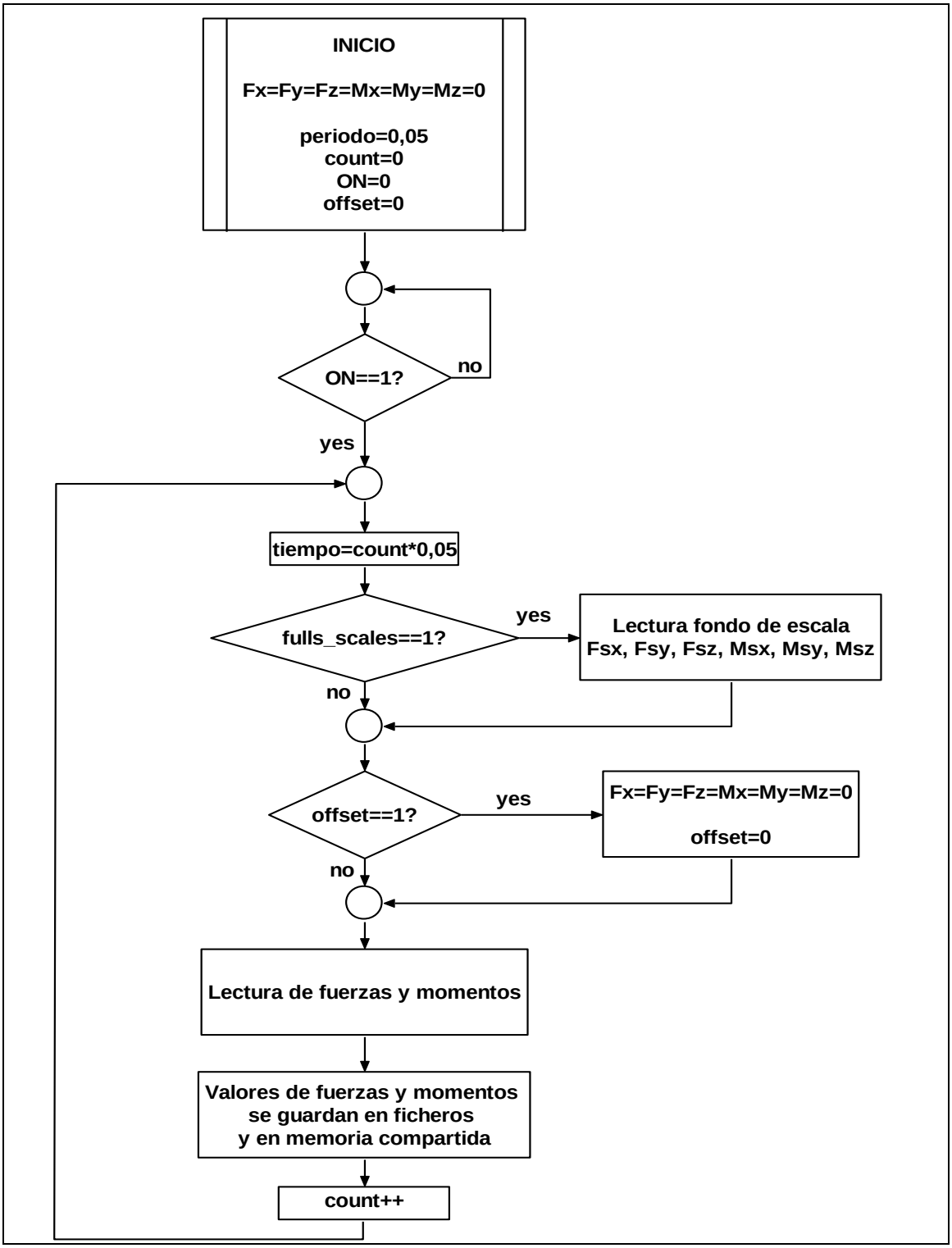


Figura 4.2. Diagrama de flujo explicativo aplicación "jr3.cpp".

a). Definición y explicación de variables utilizadas por "jr3.cpp"

Las variables utilizadas en "jr3.cpp" son:

- **FILE *fichero**: Fichero de texto donde se guardan los valores de fuerzas y momentos leídos por el sensor. Con este fichero podremos estudiar los datos tras los ensayos.
- **FILE *fichero_Matlab**: Es otro fichero de texto que contiene sólo valores numéricos (el tiempo (sg), las fuerzas(10*N) y momentos(10N*m)) separados por espacio o saltos de línea. Esto nos va facilitar el estudio a posteriori de los datos, porque con dicho formato de fichero, éste lo podremos abrir desde MATLAB y guardar todos sus datos en una matriz mediante la siguiente sentencia:

>>A=load('fichero_Matlab.txt');

Esta matriz tendrá 25 columnas, cada cual contiene la información siguiente (Tabla 4.1).

nº columna	Información	nº columna	Información
1	Tiempo	14	Fx2
2	Fx0	15	Fy2
3	Fy0	16	Fz2
4	Fz0	17	Mx2
5	Mx0	18	My2
6	My0	19	Mz2
7	Mz0	20	Fx3
8	Fx1	21	Fy3
9	Fy1	22	Fz3
10	Fz1	23	Mx3
11	Mx1	24	My3
12	My1	25	Mz3
13	Mz1		

Tabla 4.1. Información de las columnas de fichero_Matlab.

De esta forma, el estudio posteriori de los datos se simplifica, ya que con "fichero.txt" podemos saber qué valores queremos representar y teniéndolos ya guardados en la matriz A, las representaciones gráficas se simplifican una vez seleccionado qué intervalo de filas de qué columnas queremos representar:

>>plot(A(fila_inicio:fila_fin,1),A(fila_inicio:fila_fin, columna_selecionada));

Si, por ejemplo, de un ensayo queremos representar desde los 80 sg a los 120 sg una forma sencilla sería:

```
Function matriz_Matlab()
        tiempo_inicial=80;
        tiempo_final=120;
        i=1;
        while (i>o)
                if A(i,1)==tiempo_inicial
                        fila_inicio=i;
                end
                if A(i,1)==tiempo_final
                        fila_fin=i;
                        i=-1;
                end
                i = i + 0,05;
        end
```

Dividimos la columna con los datos de fuerza/par porque estos están en 10Nm

```
B=A(fila_inicio:fila_fin,columna_seleccionada);/
plot(A(fila_inicio:fila_fin,1),B);
```

Por supuesto, para poder realizar esto debemos representar desde la interfaz de MATLAB los datos respecto al tiempo de ensayo, es decir, no debemos hacer representaciones iniciando el tiempo a cero, ya que en los ficheros se inicia el tiempo

cuando se presiona ON en "interfazjr3_display". Hay que tener en cuenta que los valores de fuerza y pares se guardan en los ficheros en 10*N y 10*Nm, de ahí que dividamos los valores de la columna con los datos de fuerza o par entre 10.

- **int shmid**: Identificador del segmento de la zona de memoria obtenida a través de la función shmget().
- **static int rate=20**: Inicializamos el valor de la frecuencia de muestreo a 20Hz.
- **static int period**: Periodo de muestreo, se le asigna su valor en milisegundos:

  period = 1000000/rate;

- **float count**: Count es el número de muestra recogida. El valor de tiempo en que tomamos cada dato lo obtenemos de la expresión: count*period/1000000. La inicializamos a cero al comienzo de la ejecución del programa y después de cada recogida de datos aumentamos su valor: count=count+1.

  La PCI, como cualquier otro driver, se comunica con el espacio de usuario mediante llamadas de funciones en un fichero dispositivo (/dev/jr3). La función ioctl( ) nos permite enviar comandos específicos para un periférico, como es el caso de nuestra tarjeta de adquisición de datos. Para poder utilizarla hay que incluir en la cabecera la biblioteca <sys/ioctl.h>. Ioctl() recibe tres parámetros fd (identificador del dispositivo), luego el comando a ejecutar y un tercer parámetro optativo, un puntero donde se guardan el/los parámetros que se le han de pasar al driver o bien en donde ha de retornar éste los resultados. Por último, ioctl() nos devuelve un valor que será cero si el comando se ha ejecutado con éxito, o -1 si ha habido algún problema [14].

  En nuestro código hemos definido las siguientes variables para poder enviar comandos a la PCI a través de la función ioctl( ).

- **int ret**: Retorno de los comando enviados a la PCI mediante ioctl( ), función que devuelve (a través de ret) un valor de cero si el comando se ha ejecutado con éxito, o -1 si ha habido algún problema.

- **int fd**: Primer parámetro de sentencia ioctl ( ) para envío de comandos a la PCI. Se trata de un identificador de dispositivo. Puesto que se trata de una variable de tipo int y no de tipo FILE, el fichero del dispositivo ha de abrirse usando la funcion open( ) en lugar de fopen( ). Por la misma razón, para leer o escribir en él, ha de usarse read( ) y write( ) en vez de fread( ) o fwrite( ), y para cerrarlo se usa close( ) en vez de fclose( ).

- **six_axis_array fm0, fm1, fm2, fm3**: punteros donde la PCI ha de retornar los valores de los comandos respectivos IOCTL0_JR3_FILTER0, IOCTL1_JR3_FILTER0, IOCTL2_JR3_FILTER0, IOCTL3_JR3_FILTER0.

ret=ioctl(fd,IOCTL0_JR3_FILTER0,&fm0)

$$\begin{cases} \text{ret} = \begin{cases} 0 \text{ , comando IOCTL\_JR3\_FILTER0 éxito} \\ -1 \text{, error al ejecutar el comando IOCTL\_JR3\_FILTER0} \end{cases} \\ \text{fd:Identificador de la tarjeta PCI} \\ \text{fm0: Puntero donde se guarda los valores de FILTER0} \end{cases}$$

- **int fs0[6], fs1[6], fs2[6], fs3[6]:** Son 4 vectores de dimensión 6 donde se guardan los valores de full scales introducidos por el usuario desde la "interfazjr3_display" de Matlab. En fs0 se almacenan las full scales para el sensor0, en fs1 las correspondientes para el sensor1, en fs2 las del sensor2 y en fs3 las del sensor3.

- La conversión utilizada en las medidas dadas por el sensor consiste en aplicar a dichas medidas un factor de escala para conseguir el valor real. Esta conversión se indica en el manual del JR3 y necesita las full scales. La PCI posee 14 bits, en consecuencia, para una correcta conversión se debe multiplicar por una resolución de 1/(2^14) = 16384. Esta conversión es la que sigue:

```
mem->Fx1 =10*fm1.f[0]*fs1[0]/16384;
mem->Fy1 =10*fm1.f[1]*fs1[1]/16384;
mem->Fz1 =10*fm1.f[2]*fs1[2]/16384;
mem->Mx1 = fm1.m[0]*fs1[3]/16384;
mem->My1 = fm1.m[1]*fs1[4]/16384;
mem->Mz1 = fm1.m[2]*fs1[5]/16384;
```

En cuanto a: mem->Fx1, mem->Fy1, mem->Fz1, mem->Mx1, mem->My1, mem->Mz1, son las variable definidas en la memoria compartida "memoria.h" donde se guardan los valores respectivos de fuerzas y momentos para el sensor1. Hay otras tantas variables para el almacenamiento de los datos leídos por los otros tres sensores.

b). Diagrama de flujos de "jr3.cpp".

Una vez definidas y explicadas todas las variables podemos presentar un diagrama de flujo más detallado y técnico (Figura 4.3).

Las funciones "display_t", "write_it" y "write_Matlab" reciben de la función main( ) de "jr3.cpp" el puntero fm devuelto por la función correspondiente ioctl, así como, los fondos de escala introducidos por el usuario para casa sensor. "display.it" imprime por pantalla, tras realizar la conversión requerida, los valores de las fuerzas y momentos. Por su parte, "write_it" y "write_Matlab" también realizan la misma conversión de valores eléctricos a valores de fuerzas y momentos y guardan estos últimos en los ficheros "fichero.txt" y fichero_Matlab.txt", respectivamente. Sus diagramas de flujos explicativos, se muestran en la Figuras 4.4, Figura 4.5 y Figura 4.6

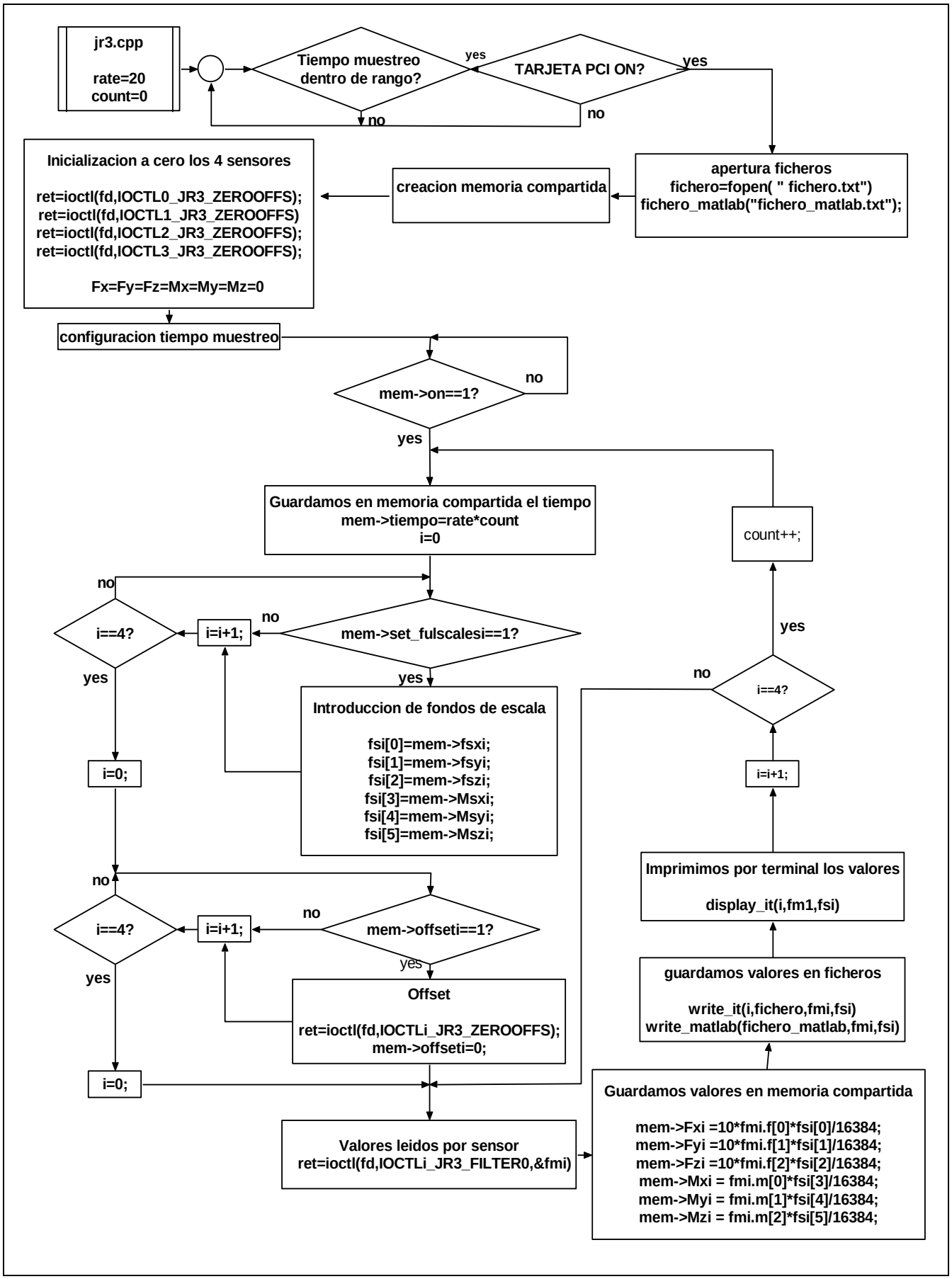


Figura 4.3. Diagrama flujo de aplicación "jr3.cpp".

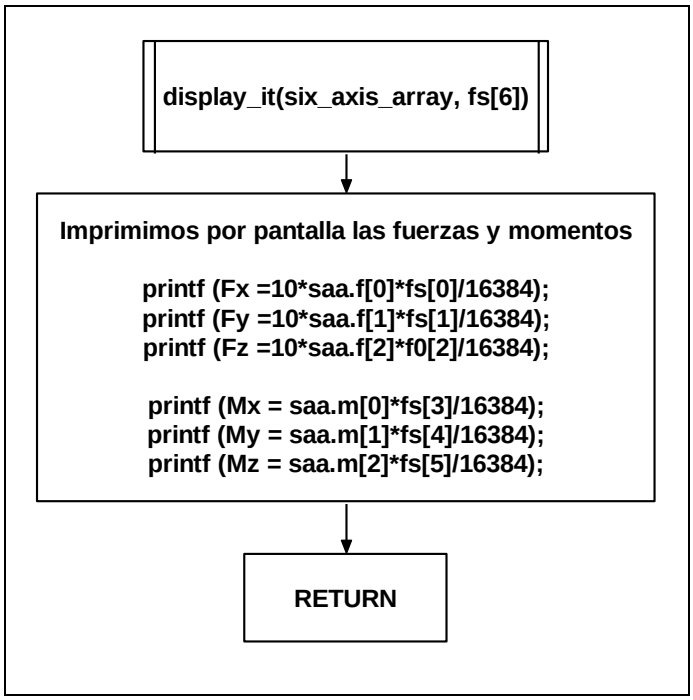


Figura 4.4. Diagrama de flujo de función "displat_it".

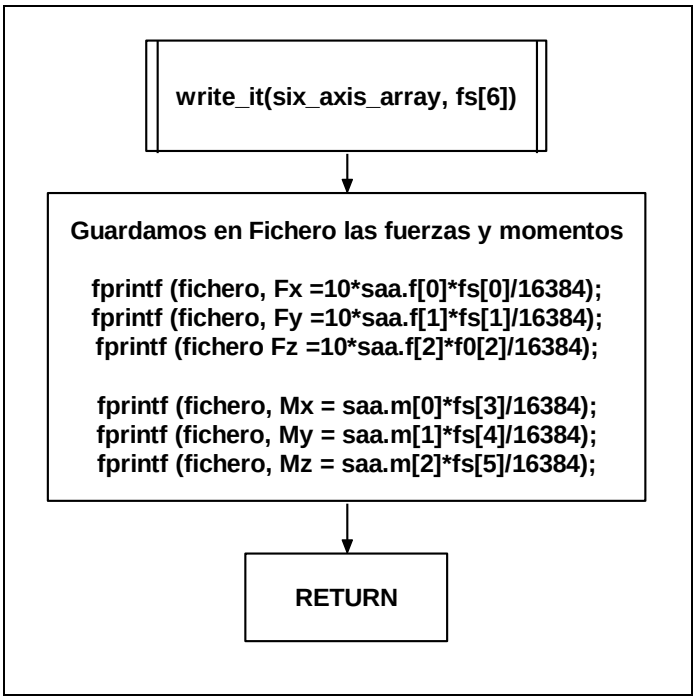


Figura 4.5. Diagrama de flujo de función "write_it".

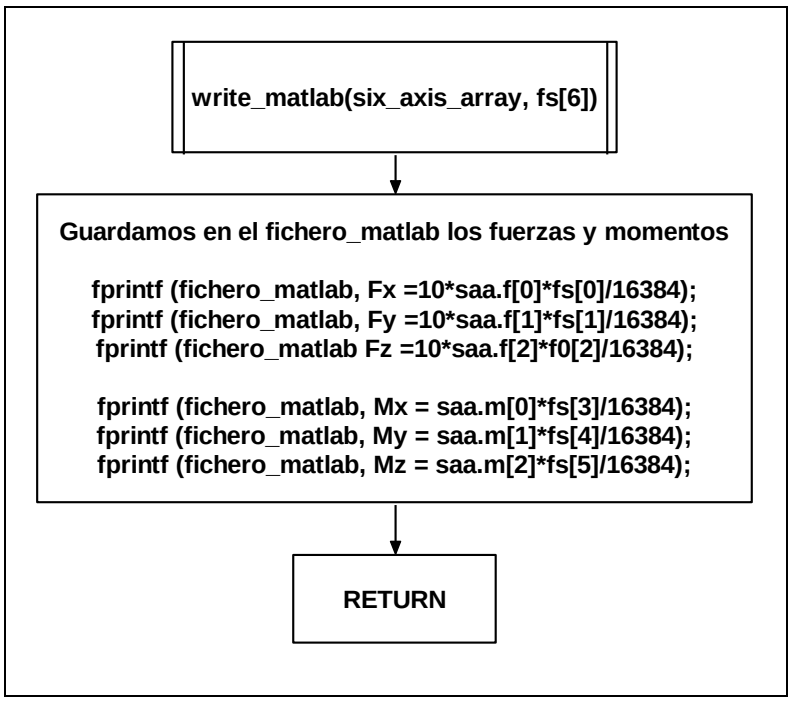


Figura 4.6. Diagrama de flujo de función "write_Matlab".

4.1.2. Aplicación "interfazjr3_display".

Para la interfaz gráfica con la que interactuará el usuario se ha utilizado Matlab Guide, que es un entorno de programación visual que ofrece Matlab para realizar y ejecutar programas de simulación a medida del usuario. Para usar esta aplicación simplemente tenemos que teclear la sentencia "guide" en la terminal de Matlab.

Guide crea dos archivos, uno .m (ejecutable, en nuestro caso "interfazjr3_display.m") y otro .fig ("interfazjr3_display.fig), la parte grafica. Las dos partes están unidas a través de subrutinas denominadas callback. Una vez que se graba los archivos desde la terminal MATLAB (si salvamos la .fig automáticamente lo hace el .m asociado y viceversa) podemos ejecutar el programa en la ventana de comando de MATLAB solamente escribiendo el nombre del archivo sin extensión, "interfazjr3_display".

El archivo .m que se crea tiene una estructura predeterminada. Consta de un

encabezado y a continuación viene el código correspondiente a las subrutinas. Cada elemento (botón) introducido en el archivo .fig le corresponde una subrutina callback en el archivo .m, de tal forma, que cuando se ejecute la interfaz y el usuario presione un botón, la subrutina del archivo .m de dicho botón se activa, por tanto, dentro de estas callback hay que implementar lo que se quiere que ejecute el botón en cuestión.

Resumidamente, nuestra interfaz nos va a permitir:

1º). Introducir en todo momento los valores de full scales para hasta cuatro sensores.

2º). Visualizar en tiempo real los valores de fuerza y momentos leídos de hasta cuatro sensores.

3º). Introducir en todo momento offset para hasta cuatro sensores.

4º). Por otro lado, no permitirá, para los cuatro sensores, realizar las representaciones gráficas respecto al tiempo de todas las fuerzas y momentos, y la representación del vector fuerza.

5º). También realizará en tiempo real el cálculo del ZMP en fase de balanceo (apoyo simple) y su posterior representación gráfica para el sensor 0 y sensor1, puertos a los que deberá conectarse los sensores de los tobillos.

La interfaz grafica diseñada se muestra en la Figura 4.7 y funciona de la siguiente manera:

Es importante que siempre que terminemos de usar la interfaz, antes de cerrar la ventana, presionemos el botón OFF.

Pasos a seguir para lectura de fuerzas/momentos:

1º). Se introducen los valores de fondo de escala de cada sensor en las casillas que se encuentran debajo de cada Full scales.

2º). Se pulsan los botones FULL SCALES para introducir los valores de fondo de escala.

3º). Y, por último, se presiona el botón ON.

Así ya se podrán observar en los displays los valores de fuerzas y pares leídos por el sensor.

4º). Presionando los botones de offset pondremos a cero las correspondientes fuerzas y momentos.

Figura 4.7. Ventana "interfazJR3_display".

Pasos a seguir para representar datos:

1º). En el cuadro "Captura de datos" seleccionamos las fuerzas y/o momentos que se quieran representar respecto al tiempo.

Vector fuerza: Representación del vector F=Fxi+Fyj+Fzk.

2º). Seleccionamos los sensores cuyas medidas queremos representar, para ello pinchamos sobre Sensor0 y/o Sensor1 y/o Sensor2 y/o Sensor3

3º). Posteriormente pinchamos sobre:

- Start sensor 0 y 1: Para inicializar toma de datos para sensor0 y sensor1
- Start sensor 2 y 3: Para inicializar toma de datos para sensor2 y sensor3

4º). Cuando queramos terminar la captura de datos a representar, pinchamos sobre:

- Finish sensor 0 y 1: Para visualizar las gráficas de sensor0 y sensor1
- Finish sensor 2 y 3: Para visualizar las gráficas de sensor2 y sensor3

Para representar la trayectoria del ZMP:

1º). Hay que introducir en el cuadro Sensor0 y/o Sensor1 la altura en milímetros a la que se encuentra instalado el sensor en la pierna del humanoide.

2º). Presionamos botón altura tobillo.

3º). Posteriormente pichamos sobre: Start sensor 0 y 1, para inicializar toma de datos de ZMP.

4º). Cuando queramos terminar la captura de datos a representar, pinchamos

sobre: Finish sensor 0 y 1, para visualizar las gráficas.

La interfaz calcula el ZMP del humanoide en fase de balanceo, es decir, en un único apoyo. Con sensor0 calculamos el punto ZMP para esa pierna en apoyo y con sensor1 con la otra en apoyo

En cada cuadro de captura de datos tenemos el botón "t inicial=0". Si lo presionamos, antes de pulsar start01 o start23, la representación gráfica tomará como origen de tiempos el 0. Si no pulsamos el botón en cuestión, la representación de los valores deseados se hará en función del tiempo de ensayo.

Hay que señalar que desde un principio se quería observar las representaciones gráficas en tiempo real al mismo tiempo que se visualizasen los valores en los displays. En una primera versión, la interfaz diseñada conseguía esto, pero ya sólo al representar una gráfica el tiempo de computo de MATLAB para dibujarla se elevaba hasta casi los 0,1sg, lo que aumentaba al doble el tiempo de muestreo, ya que la tarjeta PCI tiene una frecuencia de 20Hz, es decir nos proporciona una muestra cada 0,05 sg. Debido al cuello de botella que suponía este diseño, optamos por intentar hacer dos interfaces que trabajasen en paralelo. Sin embargo, MATLAB no nos dejaba lanzar las dos interfaces a la vez, si activamos el display las gráficas se quedaban en espera y viceversa. Así pues, ante nuestra incapacidad de conseguir interfaces en paralelo, optamos por la tercera opción que es la versión explicada en este trabajo y que muestra en tiempo real y en todo momento los valores de fuerza/pares mientras que capturamos datos, representando éstos a posteriori.

a). Definición y explicación variables “interfazJR3_display”

La aplicación “interfazJR3_display” está formada, como ya hemos explicado, por subrutinas llamadas cada vez que se pulsa su botón correspondiente. Al tratarse de subfunciones, independientes de las demás, todas las variables que definamos en los callbacks serán de tipo global, para que de esta forma si una subrutina cambia el valor

de una variable que utiliza otra, esta última reconozca dicho cambio.

Las variables definidas en nuestra interfaz son:

- **on**: Cuando se presiona su botón se empieza mostrar los valores de fuerza y pares.
- **set_fullscales0, set_fullscales1, set_fullscales2, set_fullscales3**: Si se presiona el botón con el mismo nombre se ponen a 1 y se introducen las full scales seleccionadas por el usuario.
- **set_offset0 , set_offset1, set_offset2, set_offset3**: Si se presiona el botón con el mismo nombre se pone a 1 y ponen a cero la variable correspondiente de offset (offset0, offset1, offset2, offset3).
- **offset0, offset1, offset2, offset3:** Estando a 1 se inicializan a cero los valores de fuerzas y momentos del sensor respectivo.
- **fsx0, fsy0, fsz0, Msx0, Msy0, Msz0**: Variables donde se guardan los fondos de escala introducidos por el usuario para el sensor 0.
- **fsx1, fsy1, fsz1, Msx1, Msy1, Msz1**: Idem para fondo de escala de sensor1.
- **fsx2, fsy2, fsz2, Msx2, Msy2, Msz2:** Idem para fondo de escala de sensor2.
- **fsx3, fsy3, fsz3, Msx3, Msy3, Msz3**: Idem para fondo de escala de sensor3.

  NOTA: Para los fondos de escala de los cuatro sensores las fuerzas se deben introducir en N y los momentos en 10*Nm
- **Fx0, Fy0, Fz0, Mx0, My0, Mz0**: Variables donde se guardan los valores de fuerzas y momentos leídos por el sensor conectado al puerto0 (sensor0) de la PCI.
- **Fx1, Fy1, Fz1, Mx1, My1, Mz1**: Idem para fuerzas/pares de sensor1.

- **Fx2, Fy2, Fz2, Mx2, My2, Mz2**: Idem para fuerzas/paras de sensor2.
- **Fx3, Fy3, Fz3, Mx3, My3, Mz3**: Idem para fuerzas/pares de sensor3.
- **fx0, fy0, fz0, mx0, my0, mz0, fx1, fy1, fz1, mx1, my1, mz1**: Definidas como vectores en la subfunción "captura_datos01.m", en ellas se guarda los datos de fuerzas y momentos del sensor0 y sensor1 a representar posteriormente.
- **Fx2, fy2, fz2, mx2, my2, mz2, fx3, fy3, fz3, mx3, my3, mz3**: Idem pero se definen en subfunción "captura_datos23.m" y almacenan los datos de sensor2 y sensor3
- **t01**: Se define en "captura_datos01.m" y "representación_datos01.m", es un vector donde se almacenan los valores de tiempo para representación de datos de sensor0 y sensor1.
- **t23**: Se define en "captura_datos23.m" y "representación_datos23.m", es un vector donde se almacenan los valores de tiempo para representación de datos de sensor2 y sensor3.
- **set_tini01, tini_01:** Si se presiona "tinicial=0" en el cuadro captura de datos de sensor0 y sensor1, tini_01 toma el valor del tiempo en el instante que se presiona start01 para iniciar la representación desde t=0sg.
- **set**_tini23, tini_23: Idem que anterior para sensor2 y sensor3
- **[a, b, c], [d, e, f], [g, h, k], [l, m, n]:** Vectores donde se guardan en "captura_datos" los vectores fuerza de los cuatro sensores.
- **i:** Variable para recorrer los vectores de almacenamiento definidos en "captura_datos01.m" y "representación_datos01.m".
- **j:** Variable para recorrer los vectores de almacenamiento definidos en "captura_datos23.m" y "representación_datos23.m".

- **Tiempo**: Es el tiempo de recogida de cada dato. Empieza a correr cuando se presiona ON.

- **start01, start23**: Cuando se presiona su botón se pone a 1 y empieza la captura de datos para el sensor0 y sensor1, en el caso de satrt01; y para el sensor2 y sensor3, en el de start23.

- **finish01, finish23**: Al presionar botón correspondiente, se pone a 1, y consecuentemente finaliza la captura de datos para sensor0 y sensor1 (finihs01) o para sensor2 y sensor3 (finish23).

- **pz0, pz1:** Valores en milímetros introducidos por usuario para la altura del tobillo (pz0 altura a la que se encuentra sensor0 y pz1 altura del sensor1).

- **set_pz0, set_pz1:** Si se introducen valores para pz0 y pz1, set_pz0 y set_pz1 toman el valor 1 y, en consecuencia, la interfaz calcula el ZMP para esos valores de altura.

- **(ZMPx0, ZMPy0), (ZMPx1, ZMPy1)**: Variables donde se guardan en cm las coordenadas del ZMP calculado para condiciones de sensor0 (ZMPx0, ZMPy0) y sensor1 (ZMPx1, ZMPy1)

- **zmpx0, zmpy0**: Se definen en "captura_datos01.m" y son vectores donde se guardan los valores de ZMPx0 y ZMPy0, para la posterior representación.

- **zmpx1,zmpy1**: Se definen en "captura_datos023.m" y son vectores donde se guardan los valores de ZMPx1 y ZMPy1, para su posterior representación

- **grafica_ZMP**: Para representar gráficamente el ZMP calculado con anterioridad.

- **grafica_fx01, grafica_fy01, grafica_fz01, grafica_Mx01, grafica_My01, grafica_Mz01, vector_fuerza01**: Al presionar su botón correspondiente se

ponen a 1. Al estar a 1 se grafican los parámetros correspondientes del sensor0 y/o sensor1, esta última selección depende de las variables sensor0 y sensor1.

- **grafica_fx01, grafica_fy01, grafica_fz01, grafica_Mx01, grafica_My01, grafica_Mz01, vector_fuerza01**: Idem que anterior pero para sensor2 y/o sensor3

- **sensor0, sensor1, sensor2, sensor3**: Variables asociados al botón con el mismo nombre de la interfaz. Se pone a 1 cuando se pulsa su botón y sirven para indicar que se quiere representar las gráficas del sensor correspondiente.

b). Diagramas de flujos de "interfazJR3_display"

En la Figura 4.8 se muestra el diagrama de flujo para "interfazJR3_display". La función "interfaz_display.m" llama a lo largo de su ejecución a tres subfunciones .m:

1º). "inicialización.m": Pone a cero todas las variables especificadas en el diagrama flujo de la Figura 4.9.

2º). "captura_datos01.m" y "captura_datos23.m": Estas subfunciones son llamadas desde "interfazJR3_display" si el usuario pulsa, respectivamente, en el botón start01 y/o start02. Ambas subrutinas hacen los mismo, la primera guarda los datos leídos por sensor 0 y/o sensor1, y la segunda para sensor2 y/o sensor3**.** En la Figura 4.10 y Figura 4.11, se muestran el diagrama de flujo de ambas funciones.

3º). "representacion_datos01.m" y "represenctacion_datos23.m": Estas funciones realizan las representaciones gráficas seleccionadas en la ventana de la interfaz. La primera de ellas realiza las gráficas de sensor y/o sensor1, y la segunda las de sensor2 y/o sensor3. En las Figuras 4.12 y 4.13 se muestran sus diagramas de flujos.

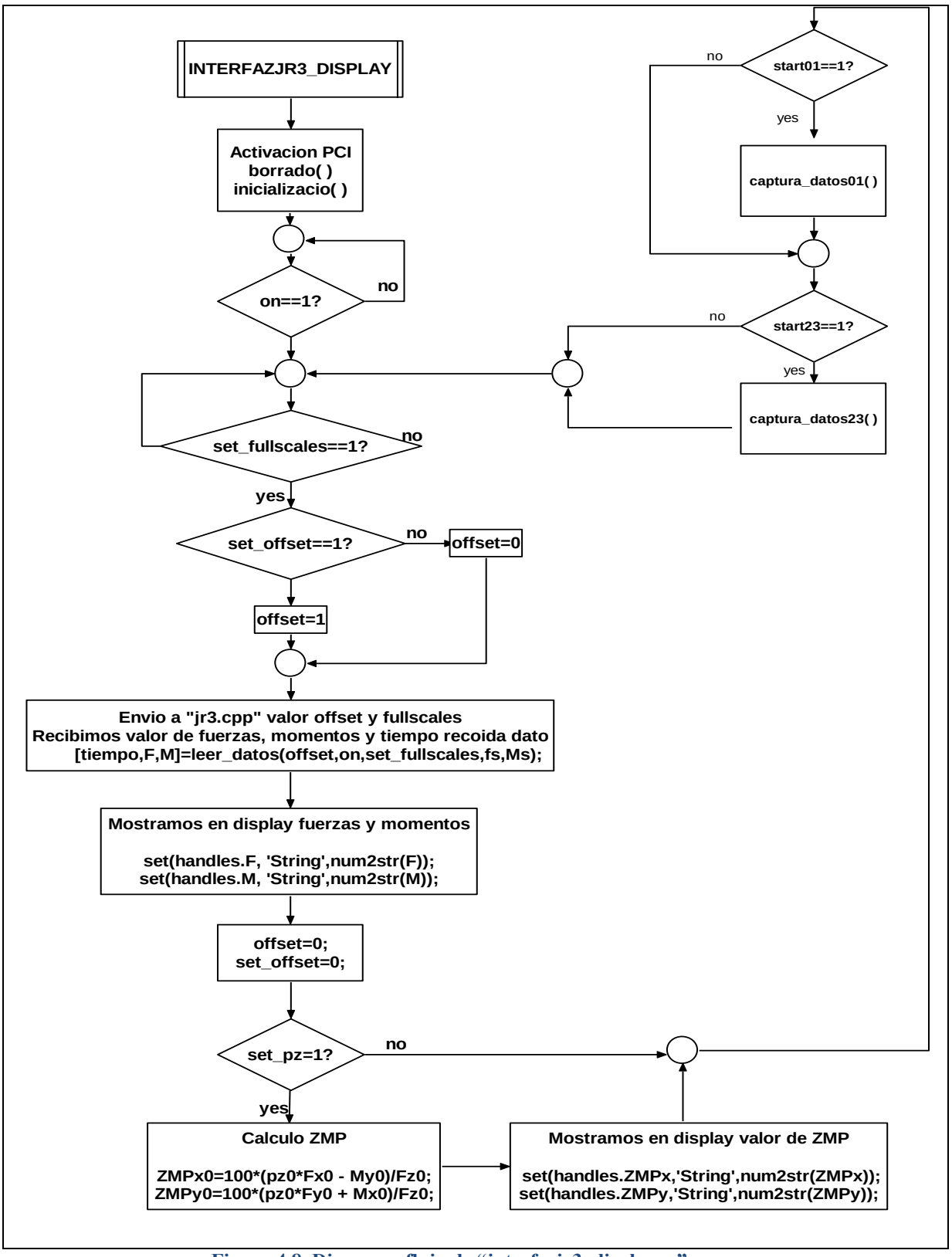


Figura 4.8. Diagrama flujo de "interfazjr3_display.m".

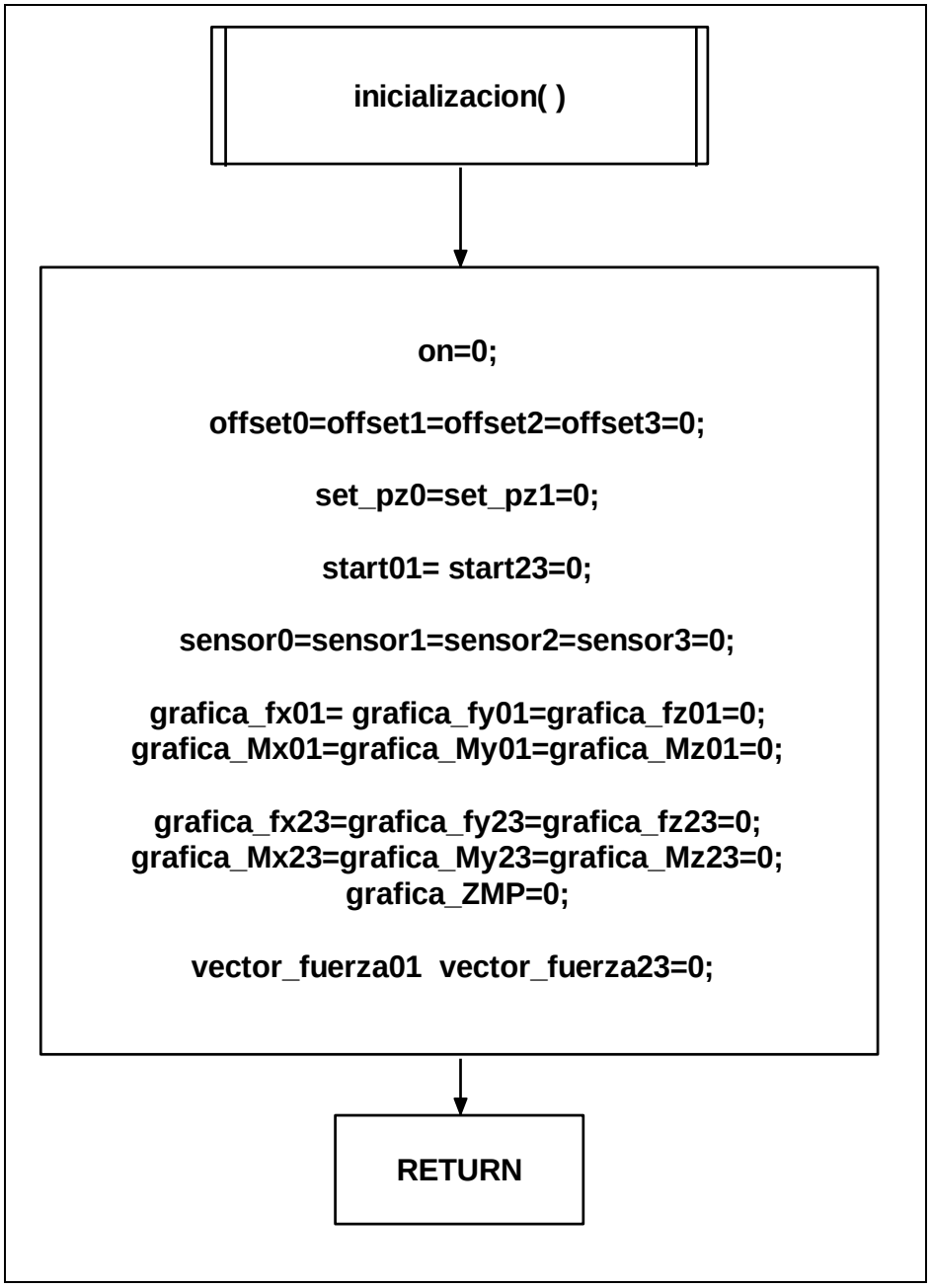


Figura 4.9. Diagrama de flujo de "inicializacion.m".

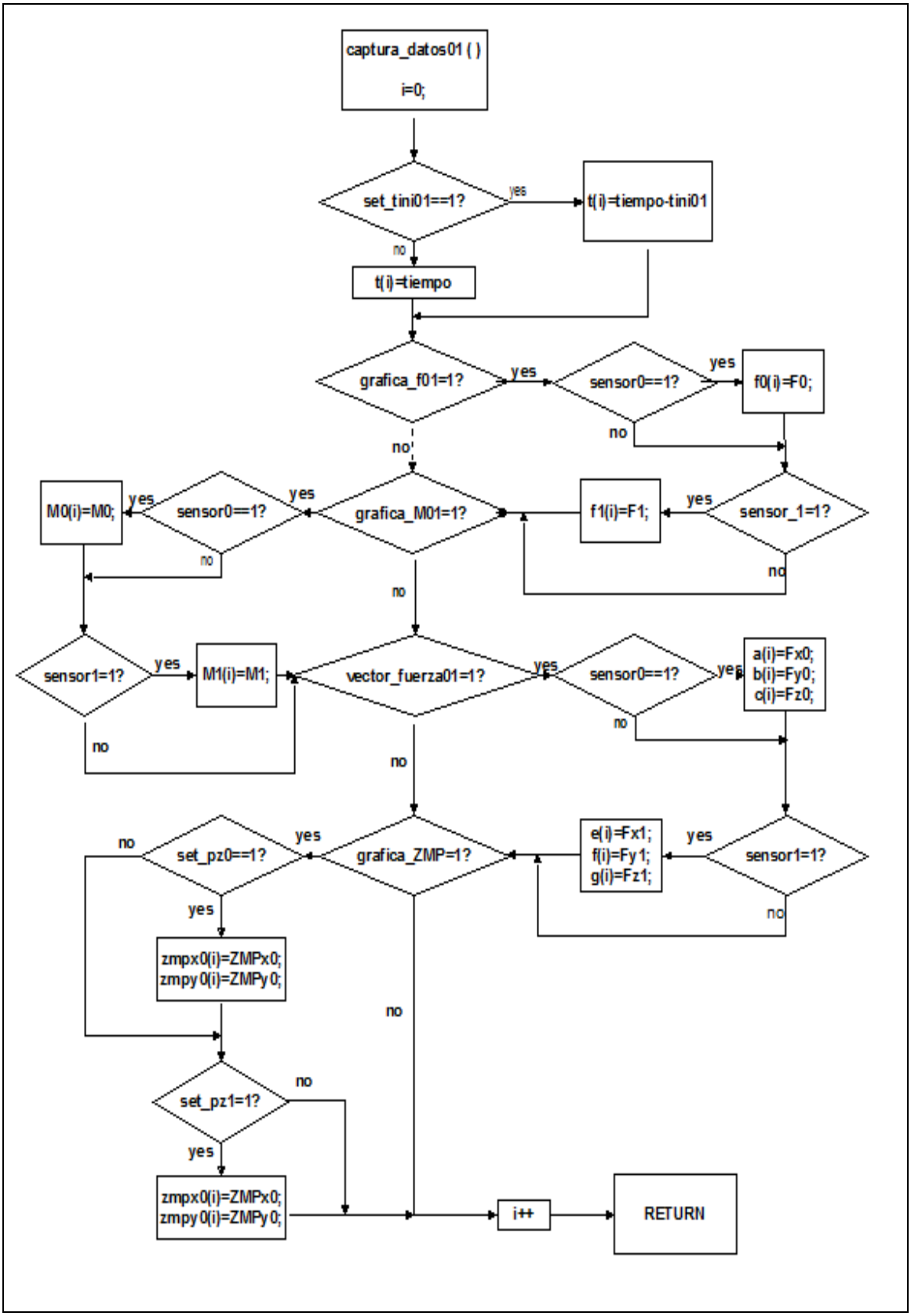


Figura 4.10. Diagrama de flujo de "captura_datos01.m".

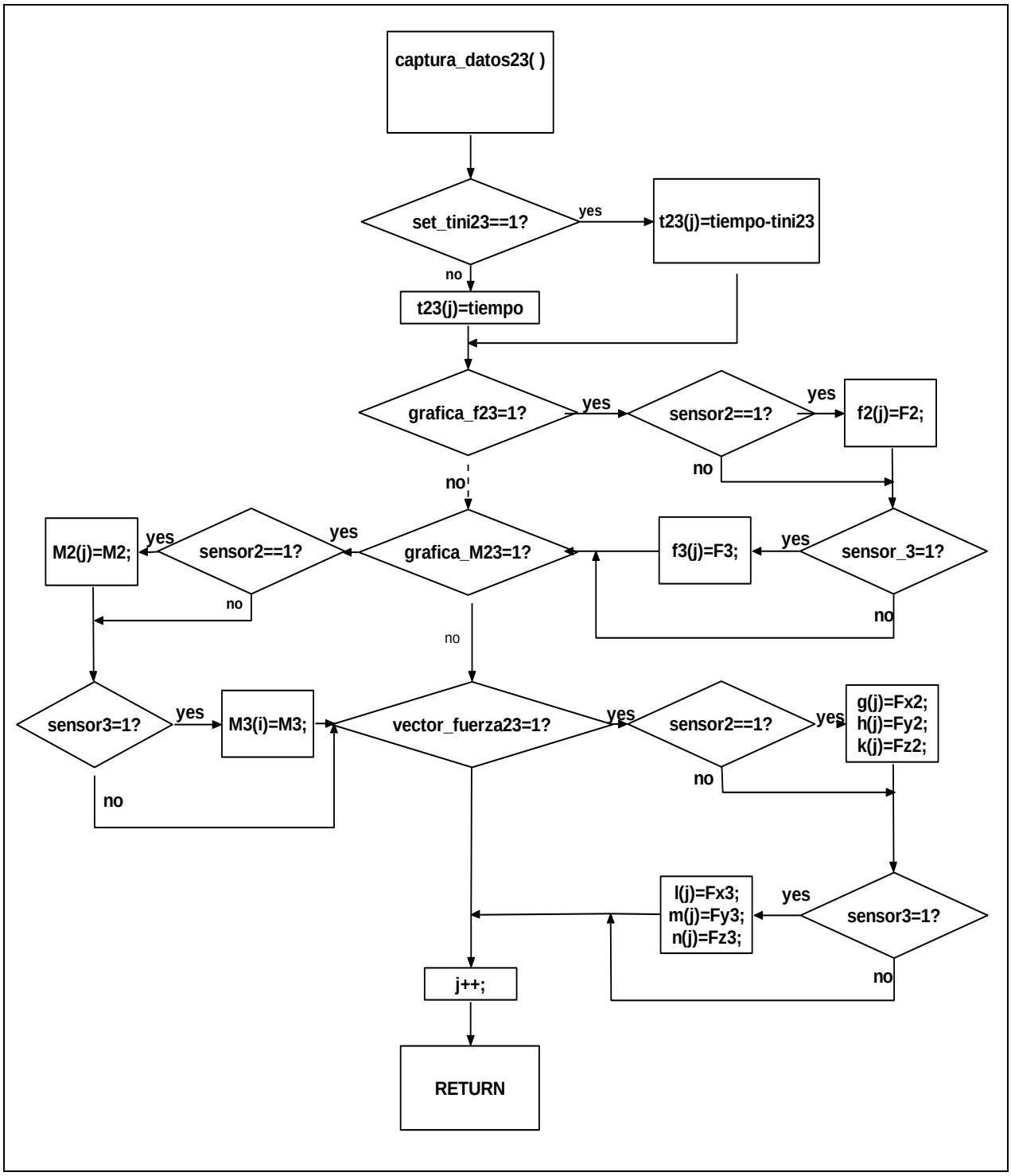


Figura 4.11. Diagrama de flujo de "captura_datos23.m".

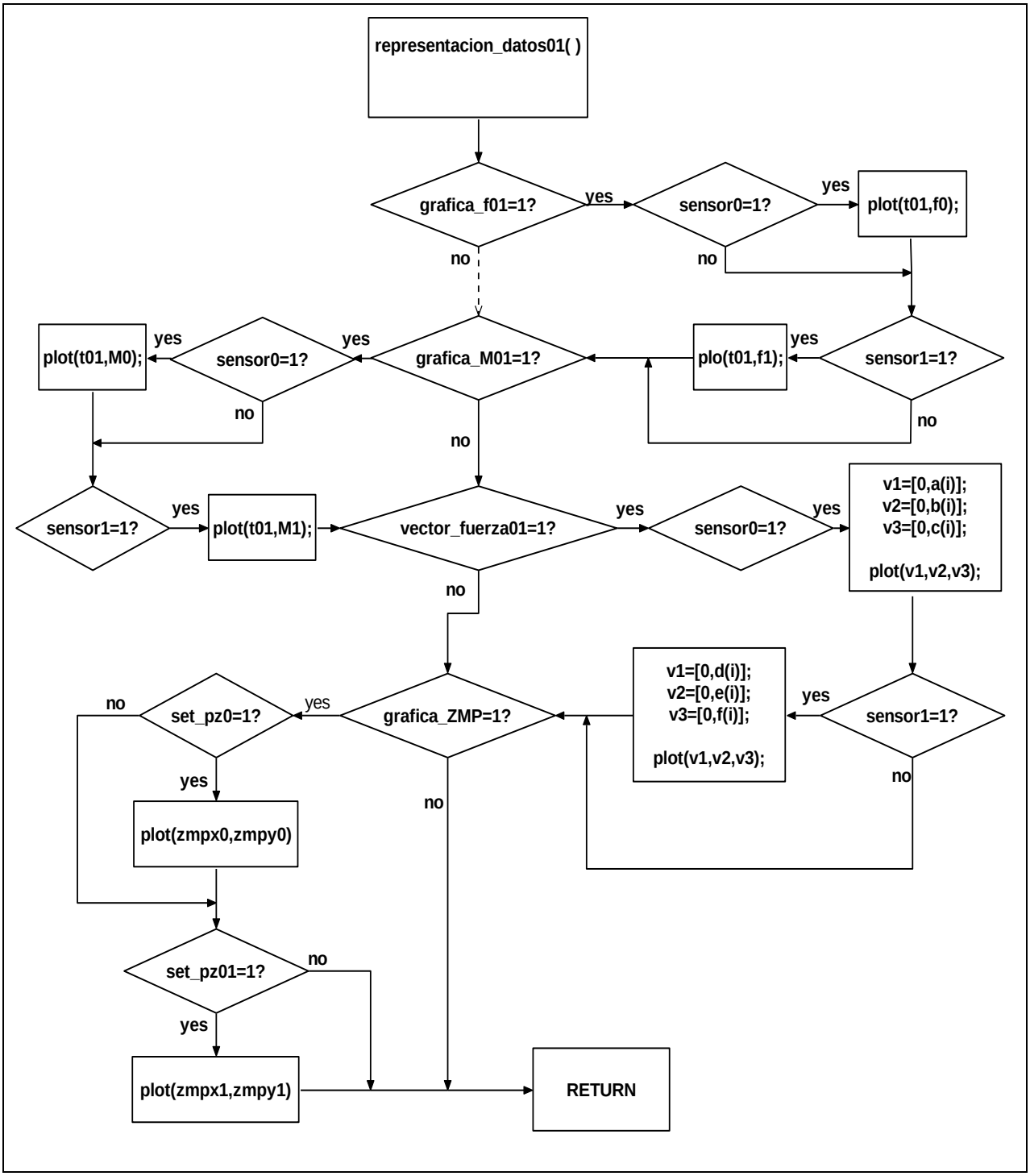


Figura 4.12. Diagrama de flujo de "representacion_datos01.m".

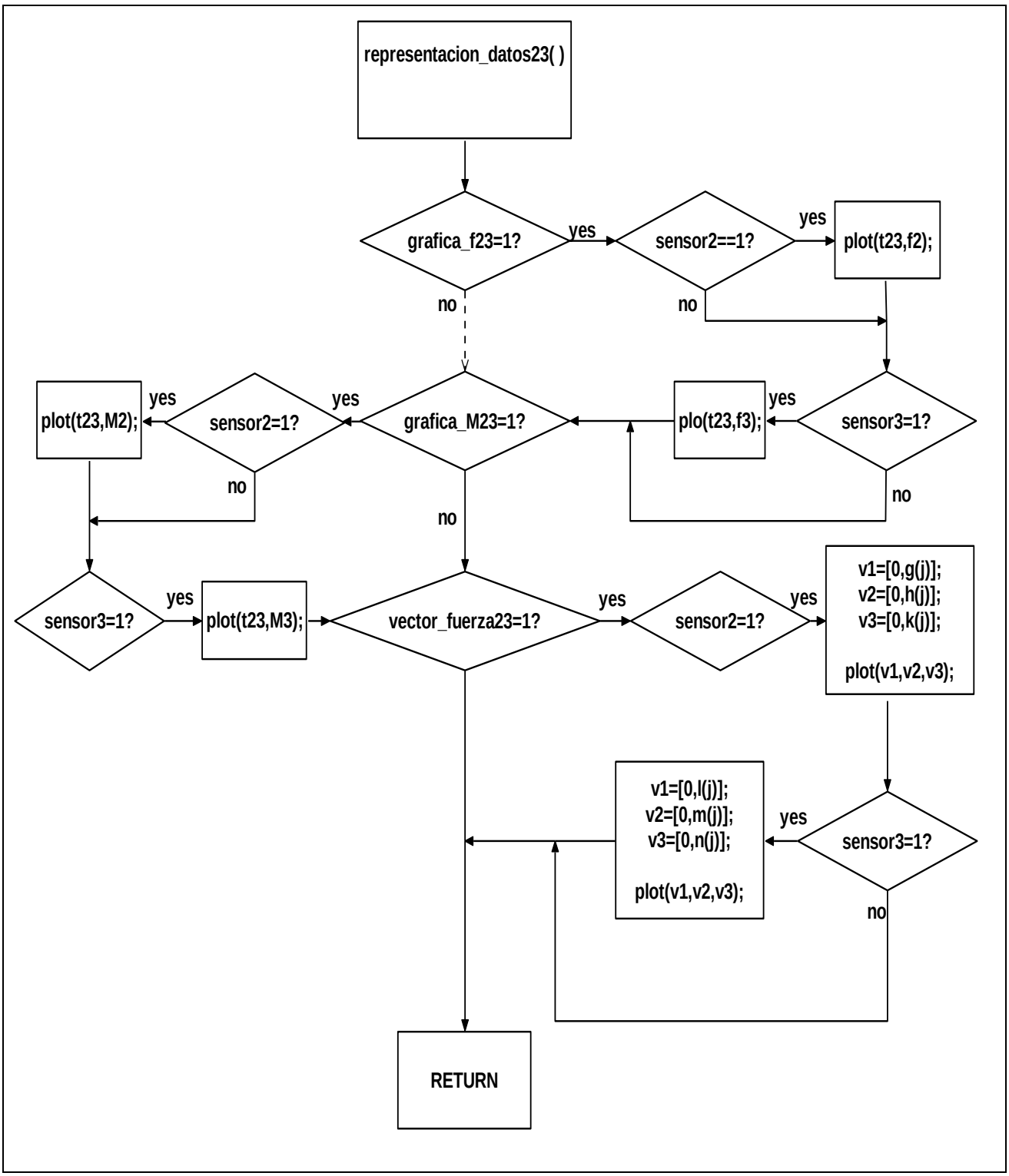


Figura 4.13. Diagrama de flujo de "respresentacion_datos23.m".

4.1.3. Aplicación de comunicación.

Las aplicaciones “jr3-cpp” e “interfazJR3_display” se comunican a través de una memoria compartida. Esta se define en el archivo “memoria.h” en el cual se obtiene la clave de acceso a la zona de memoria a compartir mediante la sentencia:

#define CLAVE_SHM ((key_t) 1001)

En este caso la clave es 1001. A continuación se definen todas las variables que van a intercambiar, entre si, las diferentes aplicaciones que tengan acceso a la memoria. De esta forma, se definen los offset, on, set_fullscales y los fondos de escala enviados por “interfazJR3_display” a “jr3.cpp”, así como, el tiempo y los valores de fuerza/pares leídos por el sensor enviados de “jr3.cpp” a “interfazJR3_display”.

La aplicación “jr3.cpp” escribe directamente en la memoria compartida. Sin embargo, la aplicación “interfazJR23_display” debe auxiliarse de funciones implementadas en C++ que ejecuta desde su código, para ello dichas funciones de comunicación deben ser de tipo MEX, las cuales son funciones ejecutables programadas en C/C++ que pueden ser cargadas y ejecutadas por MATLAB de forma automática. Nosotros hemos implementado cuatro funciones MEX “leer_datos0.cpp”, “leer_datos1.cpp”, “leer_datos2.cpp” y “leer_datos3.cpp”. Con cada una de ellas “interfazJR3_dislay” intercambia con “jr3.cpp” los datos correspondientes al número de sensor especificado en la denominación de las MEX.

Las funciones MEX tienen una extensión diferente en función de los sistemas operativos en que hayan sido generadas. En la Tabla 4.2 se puede ver la extensión que corresponde a cada sistema operativo. Nosotros utilizamos Linux, por lo que la extensión a utilizar es .mexglx [15].

Sistema operativo	Extensión del fichero MEX
Sun Solaris	.mexsol
HP-UX	.mexhpux
Linux	.mexglx
MacIntosh	.mexmac
Windows	.dll (hasta Matlab 7.0) .mexw32 (desde Matlab7.1)

Tabla 4.2. Extensiones de los ficheros MEX según sistema operativo [15].

La sintaxis del comando para compilar y crear un fichero MEX a partir de lo que se va a llamar un fichero C-MEX (un fichero C/C++ que cumple las condiciones necesarias para poder crear con él un fichero MEX) [15] se hace desde Matlab y es:

```
>> mex leer_datos0.cpp –output leer_datos0.mexglx
```

Hay diversas opciones que permiten modificar las etapas de compilación y linkado de las funciones. Las opciones principales son mostradas en la Tabla 4.3.

Opción	Función
-c	Sólo compila no linka
-g	El ejecutable incluye información para depurar la función
-O	Se optimiza el ejecutable
-outdir <name>	Se indica el directorio donde se guarda la función
-output <name>	Permite cambiar el nombre del fichero mex
- v	Se saca por pantalla cada paso del compilador

Tabla 4.3. Opciones de compilado mex [15].

El código fuente de un fichero MEX programado en C/C++ tiene dos partes. La primera parte contiene el código de la función C/C++ que se quiere implementar como fichero MEX. La segunda parte es la función mexFunction que hace de interfaz entre C /C++ y MATLAB [15].

La función mexFunction tiene cuatro argumentos: prhs, nrhs, plhs y nlhs. Estos argumentos tienen los siguientes significados:

1º). prhs es un vector de punteros a los valores de los argumentos de entrada (*right hand side arguments*) que se van a pasar a la función C/C++.

2º). nhrs es el número de argumentos de entrada de la función.

3º). plhs y **nlhs** son análogos pero referidos a los argumentos de salida (*left hand side arguments*).

Los ficheros MEX deben incluir la librería "mex.h" donde está declarada la función *mexFunction*, cuya cabecera es la siguiente:

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])

Como ya se ha dicho, los argumentos plhs y prhs son vectores de punteros a los argumentos de entrada (*right hand side*) y de salida (*left hand side*) del fichero MEX. Hay que señalar que ambos están declarados de tipo *mxArray**, es decir, que son variables de MATLAB.

En nuestro caso, los valores pasados como argumentos de entrada (prhs) desde la “inetrfazjr2_display” a las cuatro funciones “leer_datos.cpp “ son: offset, on, set-fullscales, fsx, fsy, fsz, Msx, Msy, Msz, y se guardan en la memoria con las siguiente sentencias escritas dentro de las mexFunciton:

/*se guarda en offset0 el valor de offset0 enviado desde "interfajr3_display.m"*/
offset0=(int)mxGetScalar(prhs[0]);
/*se guarda en la memoria dicho valor de offset0*/
mem->offset0=offset0;

Así se procede con todos los argumentos pasados por “interfazJR3_display” a “jr3.cpp”.

Por su parte los argumentos de salida (plhs) que recibe “interfazJR3_display” de “jr3.cpp” por medio de las diferentes funciones “leer_datos.cpp” se realiza con las siguientes sentencias dentro de las mexFunction:

```
/* Se coge de la memoria compartida el valor de tiempo guardado en la memoria por
"jr3.cpp"*/
tiempo=mem->tiempo;

/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor tiempo obtenido
de la memoria compartida*/
plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
a = mxGetPr(plhs[0]);
*(a) = tiempo;
```

Del mismo modo se hace con los demás argumentos de salida.

Por último, la manera en que “interfazJR2.display” llama desde su código a las mexFunction es igual a la llamada a cualquier función .m:

```
[tiempo, Fx0, Fy0, Fz0, Mx0, My0, Mz0] =l eer_datos0(offset0, on, set_fullscales0,
fsx0 ,fsy0 ,fsz0 ,Msx0 ,Msy0 ,Msz0);

[tiempo, Fx1, Fy1, Fz1, Mx1, My1, Mz1] = leer_datos1(offset1, on, set_fullscales1,
fsx1, fsy1, fsz1, Msx1, Msy1, Msz1);

[tiempo, Fx2, Fy2, Fz2, Mx2, My2, Mz2] = leer_datos2(offset2, on, set_fullscales2,
fsx2, fsy2, fsz2, Msx2, Msy2, Msz2);

[tiempo, Fx3, Fy3, Fz3, Mx3, My3, Mz3] = leer_datos3(offset3, on, set_fullscales3,
fsx3, fsy3, fsz3, Msx3, Msy3, Msz3);
```

4.1.4. Eliminación de la memoria compartida.

Desde la aplicación "interfazJR3_display" se elimina la memoria compartida una vez que cerramos la interfaz, momento que la interfaz llama a la función MEX borrado( ), la cual se encarga de borrar la memoria compartida mediante la función shmctl ( ):

```
int shmctl(int shmid, int cmd, struct shmid_ds *buff)
```

shmctl( ) proprociona varios modos de control sobre la memoria compartida shmid. Los modos son , entre otros:

- IPC_RMID: borrado,
- SHM_LOCK: bloquea la zona de memoria.
- SHM_UNLOCK: desbloqueo.

Como nosotros queremos borrar la memoria utilizamos el modo IPC_RMID.

Por su parte, buff es un puntero a una estructura con campos para almacenar el usuario o grupo de usuarios.

4.1.5. Aplicación para desactivar la PCI.

Otra función encargada a "interfazJR3_display" es llamar a la aplicación que desactiva la tarjeta PCI. Esta apliación es "reset.cpp" y lo único que hace es enviarle a la PCI, haciendo uso de ioctl (), el comando de desactivación (IOCTL0_JR3_RESET).

```
ret=ioctl(fd, IOCTL0_JR3_RESET);
```

4.2. Ejecución de la aplicación diseñada.

Son dos los pasos necesarios para ejecutar correctamente la aplicación diseñada:

1º). En consola de MATLAB ejecutar la interfaz:

>>“interfazJR3_display”

2º). En terminal ejecutar “jr3.cpp”:

rh-workstation:~/jr3# g++ -o jr3 jr3.c –lpthread //Primero compilamos

rh-workstation:~/jr3# ./jr3 //ejecución

“interfazJR3_display” realiza la activación la tarjeta mediante las sentencias:

%Activación de la tarjeta PCI

! make

! insmod jr3pci.ko

! make node

Antes de cerrar “interfazJR3_display” siempre debemos presionar OFF, ya que de lo contrario MATLAB nos proporcionará los siguientes errores:

??? Error using ==> set

Invalid handle object.

Error in ==> interfazJR3_display>ON_Callback at 99

set(handles.Fx0, 'String',num2str(Fx0));

Error in ==> gui_mainfcn at 96

feval(varargin{:});

Error in ==> interfazJR3_display at 18

gui_mainfcn(gui_State, varargin{:})

Error in ==>

@(hObject,eventdata)interfazJR3_display('ON_Callback',hObject,eventdata,guidata(hObject))

???? Error while evaluating uicontrol Callback

Si ahora se quiere volver a ejecutar la interfaz:

>> interfazJR3_display

Matlab nos da los siguientes errores:

??? Error using ==> isappdata

Invalid object handle

Error in ==> hgload>figload_reset at 420

if isappdata(ax,'PostDeserializeFcn')

Error in ==> hgload at 20

figload_reset(h(k));

Error in ==> openfig at 72

[fig, savedvisible] = hgload(filename, struct('Visible','off'));Error in ==> gui_mainfcn>local_openfig at 286

gui_hFigure = openfig(name, singleton, visible);

Error in ==> gui_mainfcn at 159

gui_hFigure = local_openfig(gui_State.gui_Name, gui_SingletonOpt, gui_Visible);

Error in ==> interfazJR3_display at 18

gui_mainfcn(gui_State, varargin{:});

Estos errores se deben a que no hemos presionado OFF antes de salir de la interfaz y entonces la aplicación "jr3.cpp" sigue ejecutándose y utilizando la memoria compartida. Al volver a ejecutar "interfazJR3_display", ésta va intentar lanzar un programa ya en ejecución y, en consecuencia, MATLAB dará error. Para solucionar esto, siempre que se nos olvide presionar OFF antes de cerrar la "interfazJR3_diaplay", cuando queramos volver a ejecutarla de nuevo debemos primero parar la "ejecución de "jr3.cpp", bien cerrando la terminal o bien con Ctrl+c (salida de un bucle), y posteriormente eliminar la zona de memoria compartida ejecutando desde Matlab la MEX borrado:

>>borrado

Acto seguido ya podemos ejecutar la interfaz sin ningún problema

>>interfazJR3_display

4.3. Caracterización del sensor 85M35A3-I40-DH12 de JR3 inc. en fuerzas y momentos en sus tres ejes xyz.

A continuación pasamos a estudiar las característica del sensor en sus tres ejes, para ello utilizaremos el robot ABB (Figura 4.14) que dispone la Universidad Carlos III de Madrid en su Escuela Politécnica Superior de Leganés. Se usa el ABB porque nos facilita dar al sensor (acoplado en el extremo del brazo manipulador) la orientación necesaria para situar consecutivamente los tres ejes en dirección de la gravedad, de esta forma, cuando estudiemos cada eje tendremos reducida la fuerza a su componente correspondiente al eje en cuestión.

Figura 4.14. Sensor 85M35A3-I40-DH12 de JR3 inc. instalado en robot ABB.

Hay que hacer notar que las pesas utilizadas no son patrones con calibración certificada por norma, si no que se ha optado, debido a su menor coste, por pesas de 1kg y 0.5kg sin certificado de calibrado (Figura 4.15), lo que nos introducirá, ya de por si, un error de medida, debido a que en las mismas condiciones cada pesa de 0,5kg nos dieron una fuerza vetical de 4.6 N, 4.7 N, 4,8 N y 5 N; mientras que las 8 pesas de 1 kg nos dabab lugar a una fuerza de : 9.6 N, 9.6 N, 9.7 N, 9.9 N, 9.9 N, 10 N, 10.1 N, 10.1 N. Como se puede concluir, cada una de las pesas proporcionaba un peso diferente.

Figura 4.15. Tipo de pesas utilizadas para calibración.

4.3.1. Caracterización en fuerzas.

La metodología utilizada es la que sigue:

- Orientaremos al sensor con su eje “x” como eje vertical y en este posición mediremos la fuerza en dirección “x” correspondientes a las cargas: 0, 1, 2, 3, 4, 5, 6, 7, 8 , 9, 10 kg, éste será el ciclo de subida. Posteriomente iremos midiendo las fuerzas correspondientes a cada descarga de un kilo, ciclo de bajada: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 kg . Repetiremos esta operación 5 veces, lo que supone un

total de 5 ciclos de subida y 5 ciclos de bajada. Procederemos del mismo modo con el eje "y", y posterioemente con el "z". Con todos estos datos tomados de Fx, Fy y Fz en función de la masa cargada, podremos estudiar la respuesta en fuerza del sensor en los tres ejes mediante: curva de calibración, histéresis, linealidad, exactitud y existencia de zonas muertas.

Las diferentes cargas se aplicarán, todas ellas, en el mismo punto con el fin de establecer las mismas condiciones de medida, para ello se acoplará un gancho al extremo libre del sensor del que se colgará un cubo (Figura 4.16), en el cual se irán añadiendo las pesas de 1 kg. De esta forma nos aseguramos que todas las cargas se apliquen en un mismo punto, el extremo del gancho.

Figura 4.16. Sensor fuerza/par instalado en ABB con gancho y cubo.

a). Caracterización de Fx.

Teniendo en cuenta el plano de fabricación del sensor 85M35A3-I40-DH12 de JR3 inc, adjuntado en Anexo 5, orientamos al sensor con su eje "x" en la vertical (ver Figura 4.17). Se ha utlizado un fondo de escala para Fx de 50N, por tanto el rango de medida sera de hasta 100N en el eje "x".

Figura 4.17. Orientación sistema coordenadas para calibración en Fx.

Los datos obtenidos en el ensayo se encuentran en la Tabla4.4

	Fx (N)									
Masa	**1er ciclo**		**2º ciclo**		**3er ciclo**		**4º ciclo**		**5º ciclo**	
(Kg)	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**
0	0	-0,1	0	-0,1	0	0	0	0	0	0
1	10,2	9,9	10,2	9,9	10,1	9,7	10	9,7	10	9,7
2	19,9	19,3	19,8	19,2	19,9	19,2	19,9	19,2	19,8	19,2
3	29,6	28,7	29,7	28,8	29,7	28,8	29,7	28,9	29,7	28,9
4	39,4	38,3	39,5	38,4	39,4	38,4	39,4	38,4	39,5	38,4
5	48,8	47,8	49,1	47,8	49,1	47,9	48,9	47,8	48,9	47,8
6	58,4	57,5	58,3	57,4	58,2	57,3	58,2	57,3	58,2	57,4
7	67,8	66,8	67,7	67	67,6	66,9	67,7	67	67,6	66,9
8	76,6	76,1	76,5	76	76,6	76,1	76,5	76,1	76,5	76
9	84,6	84,5	84,6	84,5	84,6	84,5	84,5	84,4	84,6	84,5
10	93,7	93,7	93,7	93,7	93,7	93,7	93,8	93,8	93,7	93,7

Tabla 4.4. Datos de calibración para Fx.

Haciendo las medias de los ciclos de subida y de bajada, obtenemos el ciclo de subida y de bajada promedios (Tabla 4.5).

Masa (Kg)	**<Fx $_{Subida}$> (N)**	**<Fx $_{Bajada}$> (N)**	**<Fx $_{Subida}$> - <Fx $_{Bajada}$> (N)**
0	0	-0,04	0,04
1	10,1	9,78	0,32
2	19,86	19,22	0,64
3	29,68	28,82	0,86
4	39,44	38,38	1,06
5	48,96	47,82	1,14
6	58,26	57,38	0,88
7	67,68	66,92	0,76
8	76,54	76,06	0,48
9	84,54	84,48	0,21
10	93,72	93,72	0

Tabla 4.5. Calibración <Fx> ciclo promedio de subida y de bajada.

A la vista de la Tabla 4.5 (columna 3) se observa una ligera diferencia entre los datos de subida y bajada, esta diferencia va aumentando a medida que aumenta la masa aplicada, llegando a su máxima diferencia (1,14N) en los 5kg (la mitad del ciclo), masa a partir de la cual la diferencia entre los ciclos vuelve a ir disminuyendo hasta converger en los 93,7 N para 10kg. Esta diferencia entre los dos ciclos nos indica que el sensor 85M35A3-I40-DH12 de JR3 inc. presenta histéresis en su medición de fuerza en el eje "x" (Fx). Hay que decir que la histéresis es directamente proporcional al fondo de escala, es decir, que se hará más intensa a medida que aumente el rango de medidas, mientras que irá desapareciendo con la disminución de dicho rango hasta llegar a un intervalo tan corto en el que el sensor no presente histéresis en sus medidas.

La histéresis lo que nos dice es que el sensor porporciona diferentes medida si una masa se carga o descarga. Ante nuestro resultados, el sensor de JR3 en estudio nos proporciona una medida de fuerza menor al descargar una masa respecto al valor medido al cargar la misma masa.

Pasamos a calcular el error de histéresis del sensor 85M35A3-I40-DH12 de JR3 inc. para Fx. Tomando de la Tabla 4.5 la máxima diferencia en fuerza entre ambos ciclos, 1,14 N para una masa de 5kg, tenemos:

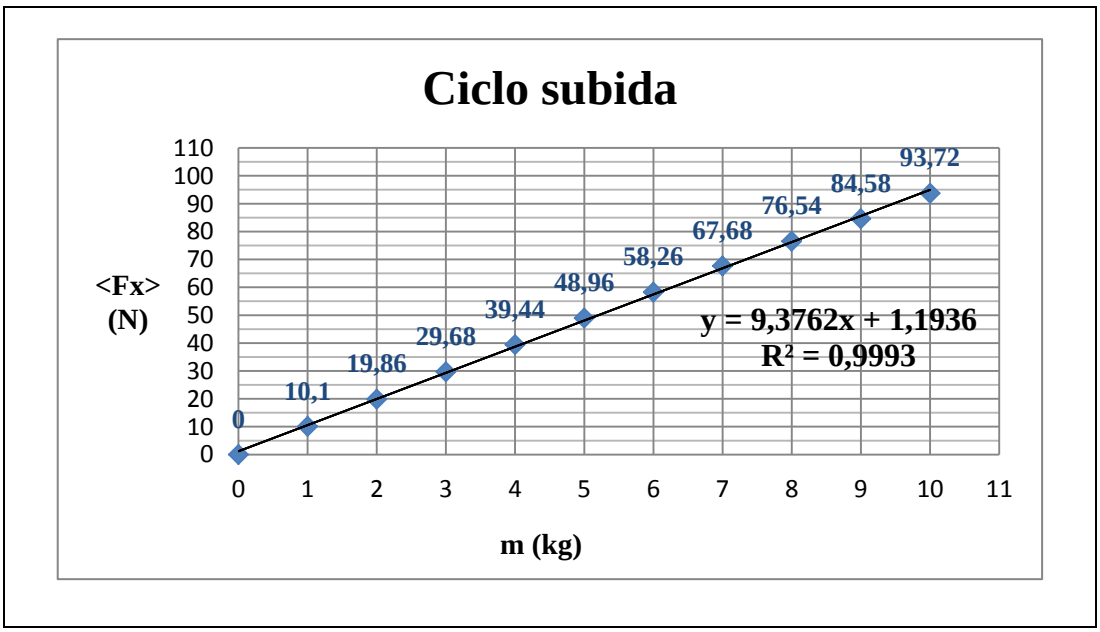
$$\varepsilon_H = \frac{\frac{\max(F_{subida} - F_{bajada})}{2}}{FS} = \frac{1{,}14/2\ N}{93{,}72\ N} = 0{,}00608 = \pm 0{,}6\%\ FS$$

$$\boldsymbol{\varepsilon_{HFx} = \pm 0{,}7\%\ FS}$$

Al presentar Fx histéresis hay que tratar sus datos de ciclo de subida independientes a los de bajada, por tanto, el sensor tendrá para la medida de fuerza en el eje "x" dos curvas de calibración, una para cada ciclo. A continuación, estudiamos por separado cada ciclo.

1º). Cálculo ciclo de calibración de ciclo de subida de Fx:

Representamos en la Gráfica 4.1 los valores promedios del ciclo de subida de la Tabla 4.5



Grafica 4.1. Curva de calibración de Fx para su ciclo de subida.

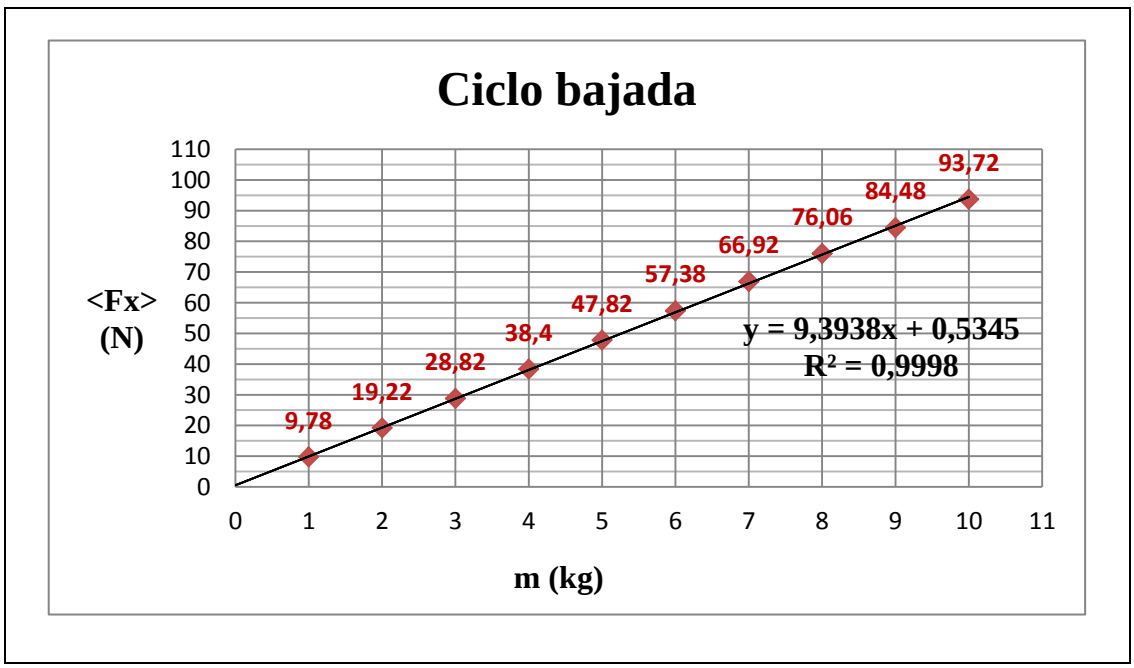
A simple vista se observa que los puntos se pueden aproximar perfectamente a una recta. Por regresión lineal, obtenemos la curva de calibración para Fx:

$$F_{x\,subida} = 9,3762 \cdot m + 1,1936 \quad \textbf{(ec. 4.1)}$$

El coeficiente de correlación lineal (r) de la expresión anterior es de $R^2 = 0,9993 \Rightarrow r = 0,9996$, lo que significa que el modelo de regresión al que hemos ajustado nuestros datos explica el 99,96% de la variabilidad de la variable Fx. Así pues, como es obvio, el modelo lineal elegido (ec. 4.1) es adecuado para describir la relación entre Fx y la masa en el correspondiente ciclo de subida.

2º). Cálculo curva de calibración ciclo de bajada de Fx:

Representado en la Gráfica 4.2 los valores promedios del ciclo de subida de la Tabla 4.5.



Grafica 4.2. Curva de calibración de Fx para su ciclo de bajada.

Para el ciclo de bajada también se observa una fuerte tendencia lineal, aplicando regresión lienal obtenemos la siguiente curva de calibración:

$$\boldsymbol{F_{x\,bajada} = 9,3938 \cdot m + 0,5345} \quad \textbf{(ec. 4.2)}$$

El coeficiente de correlación lineal (r) de la expresión anterior es de $R^2 = 0{,}9998 \Rightarrow r = 0{,}9999$, es decir, dicha recta es capaz de explicar el 99,99% de la varibilidad de Fx para su ciclo de bajada, un porcentaje muy aceptable.

En conclusión, el sensor de fuerza/par 85M35A3-I40-DH12 de JR3 inc. puede caracterizarse con las dos siguientes curvas de calibración en su medida de Fx:

$$\boldsymbol{F_{x\,subida} = 9,3762 \cdot m + 1,1936} \quad \textbf{[N]}$$

$$\boldsymbol{F_{x\,bajada} = 9,3938 \cdot m + 0,5345} \quad \textbf{[N]}$$

Una vez obtenido la curva de calibración de Fx para cada ciclo, pasamos a estudiar las características del sensor en la medida de la fuerza en eje "x":

- **Histéresis**: Como ya se ha explicado, a la vista de la Tabla 4.5, el sensor 85M35A3-I40-DH12 de JR3 inc. presenta histéresis en la medición de Fx, siendo el error introducido en la medida por dicha histéresis de:

$$\boldsymbol{\varepsilon_{HFx} = \pm 0,7\%\, FS}$$

- **Sensibilidad**: La sensibilidad de un aparato electrónico nos indica la mínima magnitud en la señal de entrada requerida para producir una determinada magnitud en la señal de salida. Matemáticamente se define como la derivada de la curva de calibración[16], es decir su pendiente,

$$\boldsymbol{S_{F_{x\,subida}} = \frac{\partial F_{x\,subida}}{\partial m} = 9,3762\,\frac{N}{kg}}$$

$$\boldsymbol{S_{F_{x\ bajada}} = \frac{\partial F_{x\ bajada}}{\partial m} = 9,3938 \frac{N}{kg}}$$

- **<u>Linealidad</u>:** La linealidad de una curva de calibración es la máxima desviación de los datos experimentales con respecto a la línea recta por la que se han aproximado [16], es decir, nos indica lo máximo que podría distanciarse el valor teórico dado por la ecuación de regresión respecto del correspondiente valor empírico.

$$\boldsymbol{\varepsilon_l = \frac{\Delta F_x}{F_{x\ max}} = \frac{max|F_{x\ real} - F_{x\ teorico}|}{F_{x\ max}}}$$

Utilizando ec. 4.1 y ec. 4.2 obtenemos para cada masa el valor teórico de Fx, con ellos podemos ver el max|Fx real - Fx teórico| (Tabla 4.6).

$$\varepsilon_{l\ F_x\ subida} = \frac{max|F_{x\ subida\ real} - F_{x\ subida\ teorico}|}{F_{x\ max}} = \frac{1,2356}{93,72} = 0,01318$$

En definitiva, el error de linealidad del sensor 85M35A3-I40-DH12 de JR3 inc. en la medida de Fx es:

$$\boldsymbol{\varepsilon_{l\ F_x} = \pm 1,3\%\ FE}$$

Masa (Kg)	Fx subida empírico (N)	Fx subida teórico (N)	Fx bajada empírico (N)	Fx bajada teórico (N)	|Fx subida empírico – Fx subida teórico | (N)	|Fx bajada empírico – Fx bajada teórico | (N)
0	0	1,1936	-0,04	0,5345	1,1936	0,5745
1	10,1	10,5698	9,78	9,9283	0,4698	0,1483
2	19,86	19,946	19,22	19,3221	0,0860	0,1021
3	29,68	29,3222	28,82	28,7159	0,3578	0,1041
4	39,44	38,6984	38,40	38,1097	0,7416	0,27
5	48,96	48,0746	47,82	47,5035	0,8854	0,3165
6	58,26	57,4508	57,38	56,8973	0,8092	0,4827
7	67,68	66,827	66,92	66,2911	0,8530	0,6289
8	76,54	76,2032	76,06	75,6849	0,3368	0,3751
9	84,58	85,5794	84,48	85,0787	0,9994	0,5987
10	93,72	94,9556	93,72	94,4725	1,2356	0,7525
		max| Fx real – Fx teorico|			**1,2356**	

Tabla 4.6. Cálculo de error de linealidad para Fx.

- **Exactitud:** Es el grado de concordancia entre el valor medio obtenido de una serie de resultados y el valor verdadero o el aceptado como referencia.

$$\varepsilon_{exactitud} = \frac{max(|valor\ medio - valor\ real|)}{valor\ medio\ maximo}$$

En la Tabla 4.7 se muestran los valores de Fx para ciclo de subida y de bajada y el correspondiente valor teórico para cada masa utilizando un valor de gravedad de 9,8m/sg.

Masa (kg)	$F_{teórico}$ con g=9,8	$<F_{xsubida}>$ (N)	$<F_{xbajada}>$ (N)	$\|F_{teórico}-F_{x\ subida}\|$ (N)	$\|F_{teórico}-F_{xbajada}\|$ (N)
0	0	0	-0,04	0	0,04
1	9,8	10,1	9,78	0,3	0,02
2	19,6	19,86	19,22	0,26	0,38
3	29,4	29,68	28,82	0,28	0,58
4	39,2	39,44	38,38	0,24	0,82
5	49	48,96	47,82	0,04	1,18
6	58,8	56,26	57,38	2,54	1,42
7	68,6	67,68	66,92	0,92	1,68
8	78,4	76,54	76,06	1,86	2,34
9	88,2	84,54	84,48	3,66	3,72
10	98	93,72	93,72	4,28	4,28
		Max(|Fteorico-Fx|)		**4,28**	

Tabla 4.7. Cálculo excatitud Fx.

Tomando de la Tabla 4.7 la máxima deifrencia entre la valore de refrencia y el experimental obtenemos un error de excatitud apar Fx de:

$$\varepsilon_{exactitud\ Fx} = \frac{4{,}28}{93{,}72} = 0{,}04567$$

$$\boldsymbol{\varepsilon_{exactitud\ Fx} = 4{,}6\%FE}$$

- **<u>Zona muerta</u>:** Se denomina así a aquellos intervalos de carga que producen una misma salida en el sensor, es decir, las regiones de la curva de calibración que presentan una sensibilidad nula [16]. Como se observa en la Grafica 4.2 y la Gráfica 4.3, ni la curva de calibración del ciclo de subida ni la del ciclo bajada

presentan zonas muertas. Por tanto, el sensor 85M35A3-I40-DH12 de JR3 inc. no presenta zonas muertas en su medida de la fuerza en el eje "x".

b). Carecterización de Fy

Situamos al sensor en la posición de la Figura 4.18.

Figura 4.18. Orientación sistema coordenadas para calibración en Fy.

El fondo de escala utilizado para la media de fuerza en el eje "y" ha sido de 55N, lo que supone un rango de medida de hasta unos110 N. La Tabla 4.8 contiene los datos tomados para la medida de Fy.

	Fy (N)									
Masa (Kg)	**1er ciclo**		**2º ciclo**		**3er ciclo**		**4º ciclo**		**5º ciclo**	
	Subida	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**
0	0	0	0	0	0	-0,1	0	-0,1	0	-0,1
1	10,4	10	10,4	10	10,5	9,9	10,5	9,9	10,5	10
2	20,1	19,8	20,3	19,7	20,3	19,9	20,4	19,9	20,4	20
3	29,8	29,1	29,7	29,3	29,9	29,3	29,9	29,3	29,9	29,4
4	39,3	38,2	39,1	38,1	39,2	38,2	39,3	38,2	39,1	38,1
5	48,4	47,3	48,4	47,3	48,5	47,4	48,5	47,5	48,5	47,4
6	57,1	56,6	57,1	56,5	57,2	56,6	57,1	56,6	57,1	56,5
7	66,3	65,7	66,4	65,9	66,2	65,8	66,3	65,8	66,2	65,7
8	75,5	75,2	75,4	75,1	75,4	75	75,5	75	75,4	75,1
9	84,5	83,9	84,4	84,1	84,4	84,2	84,5	84,2	84,5	84,2
10	92,9	92,9	93	93	93	93	93	93	92,9	92,9

Tabla 4.8. Datos de calibración para Fy.

Haciendo las medias respectivas para ciclo de subida y el ciclo de bajada, y, acto seguido, calculando sus diferencias de fuerza para cada masa tenemos los datos de la Tabla 4.9.

Masa (Kg)	**<Fy $_{Subida}$> (N)**	**<Fy $_{Bajada}$> (N)**	**<Fy $_{Subida}$> - <Fy $_{Bajada}$> (N)**
0	0	-0,02	0,02
1	10,46	9,98	0,5
2	20,3	19,86	0,44
3	29,84	29,28	0,56
4	39,2	38,16	1,04
5	48,46	47,4	1,06
6	57,12	56,56	0,56
7	66,28	65,78	0,5
8	75,44	75,08	0,36
9	84,46	84,12	0,34
10	92,96	92,96	0

Tabla 4.9. Calibración <Fy> ciclo promedio de subida y de bajada.

Podemos observar de la Tabla 4.9 como, al igual que ocurría para la medida de fuerza en el eje "x", el sensor en estudio presenta también histéresis en su medición de fuerza en el eje "y". De la tercera columna de la Tabla 4.9, vemos como los valores de fuerza correspondientes a cada ciclo se van diferenciando entre si a medida que aumenta la masa, manteniéndose el ciclo de bajada siempre por debajo al de subida. La diferencia máxima entre los dos ciclos se alcanza en el mitad de ciclo (5kg) con una diferencia de 1,06 N, a los lados de esta masa la diferencia entre ciclo se estrecha hasta llegar ambos a los 92,96 N para 10 kg y los 0 N para 0 kg.

El error correspondiente a la histéresis del sensor en Fy lo calculamos tomando de la Tabla 4.9 la máxima diferencia en fuerza de ambos ciclos, la cual es, como ya hemos comentado, 1,06 N para una masa de 5kg:

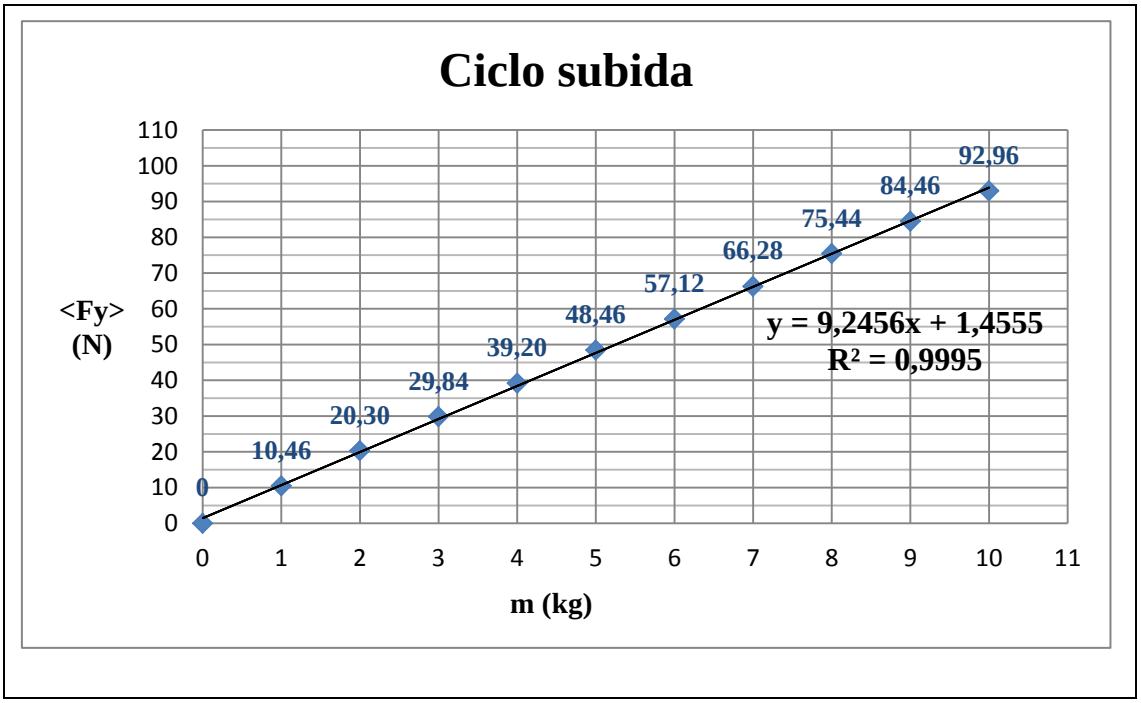
$$\varepsilon_H = \frac{\max\left(F_{ysubida} - F_{ybajada}\right)/2}{FS} = \frac{1{,}06/2\ N}{92{,}96\ N} = 0{,}0057 = \pm 0{,}6\%\ FS$$

$$\boldsymbol{\varepsilon_{HFy} = \pm 0{,}6\%\ FS}$$

A contincuación calculamos las dos curvas de calibración que caracterizan a nuestro sensor de fuerza/par en su medición de fuerza en el eje "y".

1º). Cálculo de la curva de calibración para el ciclo de subida de Fy:

Representamos los puntos del ciclo promedio de subida de la Tabla 4.9 (Grafica 4.3).



Grafica 4.3. Curva de calibración de Fy para su ciclo de subida.

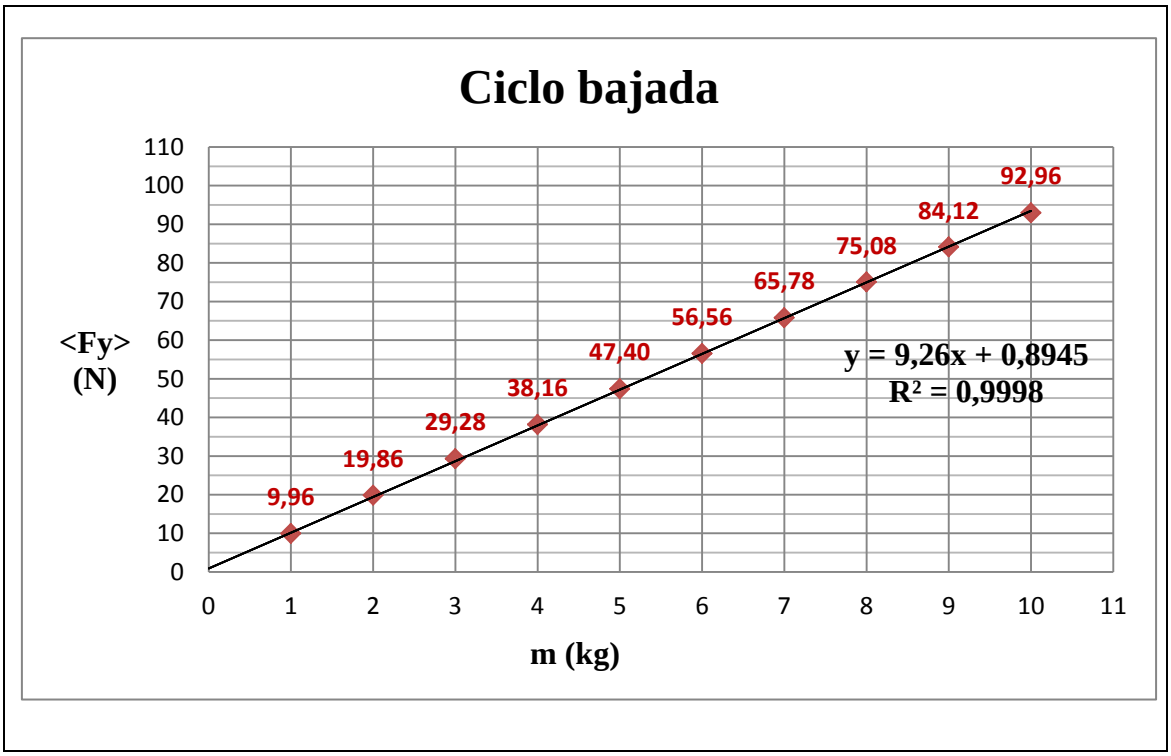
Los puntos los podemos aproximar por regresión lineal a la recta:

$$F_{y\,subida} = 9,2456 \cdot m + 1,4555 \qquad \textbf{(ec. 4.3)}$$

Aproximación con un coeficiente de correlación lineal de prácticamente 1, exactamene, r=0,9995, lo que indica que dicha aproximación explica 99,97 de la variabilidad de la variable Fy con respecto a la masa aplicada.

2º). Cálculo curva de calibración ciclo de bajada de Fy:

Representamos el ciclo promedio de bajada para Fy (Tabla 4.9) en la Gráfica 4.4:



Grafica 4.4. Curva de calibración de Fy para su ciclo de bajada.

Mediante regresión lineal se obtiene la curva de aproximación dibujada en la Grafiica 4.4:

$$F_{y\,bajada} = 9,26 \cdot m + 0,8945 \qquad \textbf{(ec 4.4)}$$

El coeficiente de correlación lineal (r) de la expresión anterior es de $R^2 = 0,9998 \Rightarrow r = 0,9999$, esto es, dicha recta es capaz de explicar el 99,99% de la varibilidad de Fy respecto a la descarga de carga.

Por tanto, el sensor 85M35A3-I40-DH12 de JR3 inc. puede caracterizarse con las dos siguientes curvas de calibración en su medición de fuerza en el eje "y":

$$F_{y\,subida} = 9,2456 \cdot m + 1,4555 \quad [N]$$

$$F_{y\,bajada} = 9,26 \cdot m + 0,8945 \quad [N]$$

Una vez obtenido la curva de calibración de Fy para cada ciclo, vamos a estudiar las características del sensor en la medida de Fy.

- **Histéresis**: Como ya se ha explicado, a la vista de la Tabla 4.9, el sensor 85M35A3-I40-DH12 de JR3 inc. presenta histéresis en la medición de Fy, siendo el error introducido en la medida por dicha histéresis de:

$$\varepsilon_{HFy} = \pm 0,6\%\, FS$$

- **Sensibilidad**:

$$S_{F_{y\,subida}} = \frac{\partial F_{y\,subida}}{\partial m} = 9,2456\ \frac{N}{kg}$$

$$S_{F_{y\,bajada}} = \frac{\partial F_{y\,bajada}}{\partial m} = 9,26\ \frac{N}{kg}$$

- **Linealidad:**

$$\varepsilon_l = \frac{\Delta F_y}{F_{y\,max}} = \frac{max|F_{y\,real} - F_{y\,teorico}|}{F_{y\,max}}$$

Utilizando la ecuaciones correspondientes a las curvas de calibración de Fy (ec. 4.3 y ec. 4.4) obtenemos para cada masa el valor teórico de Fy, con ellos podemos hallar el max|Fy real - Fy teórico| (Tabla 4.10).

Masa (Kg)	Fy subida empírico (N)	Fy subida teórico (N)	Fy bajada empírico (N)	Fy bajada teórico (N)	|Fy subida empírico – Fy subida teórico | (N)	|Fx bajada empírico – Fy bajada teórico | (N)
0	0	1,4555	-002	0,8945	1,4555	0,9145
1	10,46	10,7011	998	10,1545	0,2411	0,1945
2	20,30	19,9467	19,86	19,4145	0,3533	0,4455
3	29,84	29,1923	29,28	28,7645	0,6477	0,6055
4	39,20	38,4379	38,16	37,9345	0,7621	0,2255
5	48,46	47,6835	47,40	47,1945	0,7765	0,2055
6	57,12	56,9291	56,56	56,4545	0,1909	0,1055
7	66,28	66,1747	65,78	65,7145	0,1053	0,0655
8	75,44	75,4203	75.08	74,9745	0,0197	0,1055
9	84,46	84,6659	84,12	84,2345	0,2059	0,1145
10	92,96	93,115	92,96	93,4945	0,9515	0,5345
		max| Fy real – Fy teorico|			**1,4555**	

Tabla 4.10. Cálculo de error de linealidad para Fy.

$$\varepsilon_{l\,F_y} = \frac{max|F_{y\ real} - F_{y\,s\,teorico}|}{F_{y\,max}} = \frac{1{,}4555}{92{,}96} = 0{,}0156$$

En definitiva, el error de linealidad del sensor 85M35A3-I40-DH12 de JR3 inc. en la medida de Fy es:

$$\boldsymbol{\varepsilon_{l\,F_y} = \pm 1{,}6\%\ FE}$$

- **Exactitud**: En la Tabla 4.11 se muestran los valores de Fy para ciclo de subida y de bajada y el correspondiente valor teórico para cada masa utilizando un valor de gravedad de 9,8m/sg.

Masa (kg)	**$F_{teórico}$ con g=9,8**	**$<F_{ysubida}>$ (N)**	**$<F_{ybajada}>$ (N)**	**$\|F_{teórico}-F_{y\ subida}\|$ (N)**	**$\|F_{teórico}-F_{y\ bajada}\|$ (N)**
0	0	0	-0,02	0	0,02
1	9,8	10,46	9,98	0,66	0,18
2	19,6	20,3	19,86	0,7	0,26
3	29,4	29,84	29,28	0,44	0,12
4	39,2	39,2	38,16	0	1,04
5	49	48,46	47,4	0,54	1,6
6	58,8	57,12	56,56	1,68	2,24
7	68,6	66,28	65,78	2,32	2,82
8	78,4	75,44	75,08	2,96	3,32
9	88,2	84,46	84,12	3,74	4,08
10	98	92,96	92,96	5,04	5,04
		Max(|Fyteorico-Fy|)		**5,04**	

Tabla 4.11. Cálculo excatitud Fy.

Tomando la máxima diferencia respecto al valor referencia el valor refrencia:

$$\varepsilon_{exactitud} = \frac{max(|valor\ medio - valor\ real|)}{valor\ medio\ maximo}$$

$$\varepsilon_{exactitud\ Fy} = \frac{5{,}04}{92{,}96} 0{,}0542$$

$$\boldsymbol{\varepsilon_{exactitud\ Fy} = 5{,}4\%FE}$$

- **<u>Zona muerta</u>:** A la vista de la Gráfica 4.3 y Gráfica 4.4, las curvas de calibración para Fy no presentan zonas en las que varias cargas den lugar al mismo valor de fuerza, es decir, no presenta zonas muertas.

c). Caracterización de Fz

Para que la fuerza medida por el sensor sea únicamente la correspondiente a su componente "z" tomamos medidas (Tabla 4.12) con la configuración de la Figura 4.19.

Figura 4.19. Orientación sistema coordenadas para calibración en Fz.

Masa (Kg)	Fz (N)									
	1er ciclo		2º ciclo		3er ciclo		4º ciclo		5º ciclo	
	Subida	Bajada	Subida	Bajada	Subida	Bajada	Subida	Bajada	Subida	Bajada
0	0	0	0	0	0	0	0	0	0	0
1	9,9	9,8	9,9	9,9	10	9,8	9,8	9,9	10	9,8
2	19,9	19,9	20	20,1	20,1	20	19,9	20	20	20,1
3	29,8	29,8	29,9	29,8	29,8	30	29,8	29,8	29,8	29,9
4	39,7	39,7	39,7	39,6	39,7	39,7	39,6	39,5	39,5	39,7
5	49,5	49,6	49,6	49,5	49,4	49,5	49,5	49,6	49,5	49,7
6	59,3	59,4	59,4	59,2	59,3	59,4	59,3	59,3	59,2	59,4
7	69	69,1	69,1	68,9	69,1	69,1	68,9	69	68,9	69
8	78,8	78,9	78,9	78,8	78,8	78,9	78,9	79	78,9	78,8
9	88,8	88,8	88,9	88,8	88.9	88,9	88,9	88,8	88,9	88,9
10	97,7	97,7	97,7	97,7	97,6	97,6	97,7	97,7	97,7	97,7

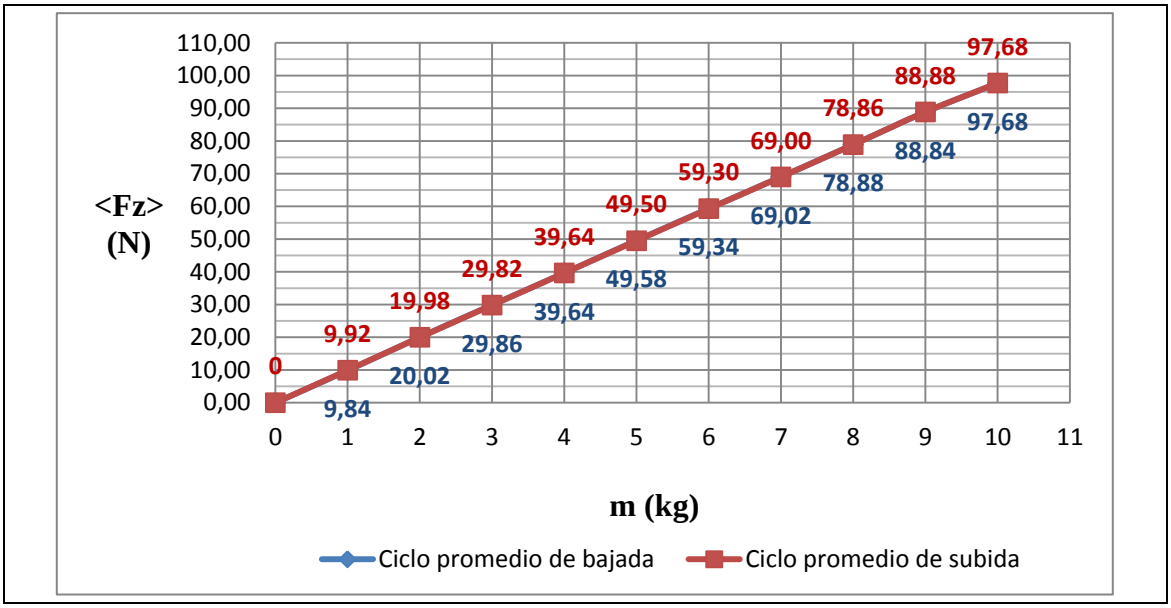
Tabla 4.12. Datos de calibración para Fz.

Presentamos en la siguiente tabla (Tabla 4.13) los valores correspondientes a la media del ciclo de subida y bajada para Fz.

Masa (Kg)	$\langle Fz_{Subida} \rangle$ (N)	$\langle Fz_{Bajada} \rangle$ (N)	$\langle Fz_{Subida} \rangle - \langle Fz_{Bajada} \rangle$ (N)
0	0	0	0
1	9,92	9,84	0,08
2	19,98	20,02	-0,04
3	29,82	29,86	-0,04
4	39,64	39,64	0
5	49,50	49,58	-0,08
6	59,30	59,34	-0,04
7	69,00	69,02	-0,02
8	78,86	78,88	-0,02
9	88,88	88,84	0,03
10	97,68	97,68	0

Tabla 4.13. Calibración <Fz> ciclo promedio de subida y de bajada.

Claramente, se observa en la Tabla 4.13 como no existe histéresis en las medidas tomadas para Fz. Al representar los ciclos promedios de subida y bajada estos quedan totalmente solapados (Gráfica 4.5).



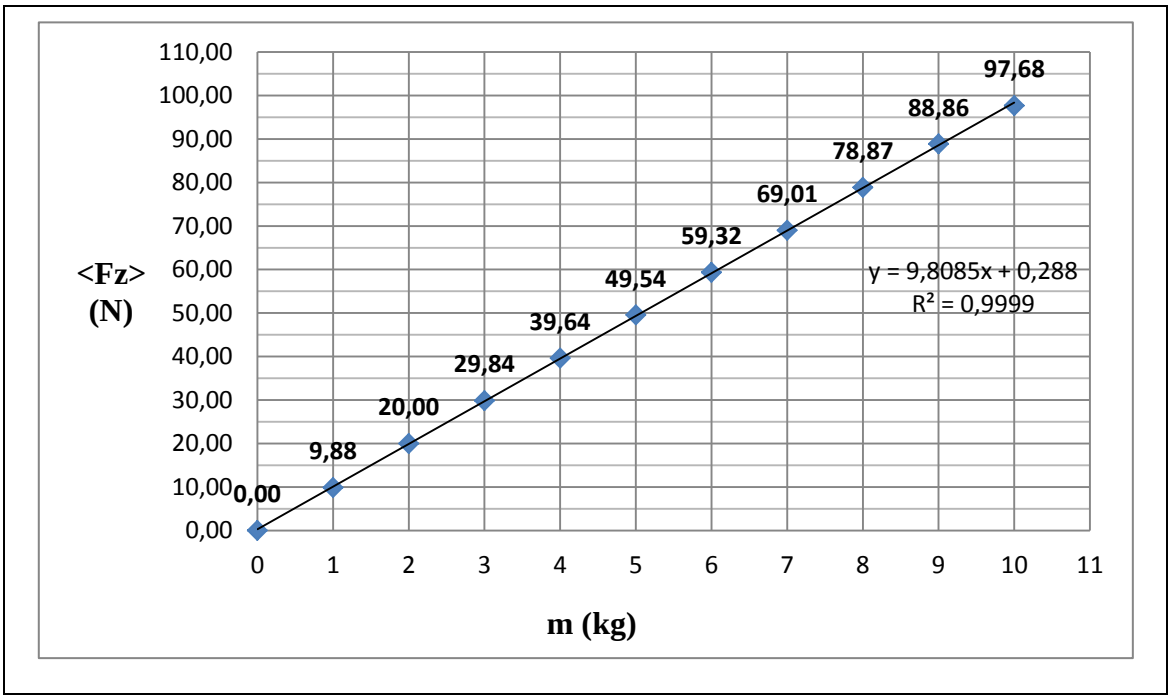
Grafica 4.5. <Fz> Vs m: Ciclo promedio de subida y de bajada.

Al no presentar el sensor 85M35A3-I40-DH12 de JR3 inc. histéresis en su medición de fuerza a lo largo del eje "z", tendrá una única curva de calibración que explicará tanto los ciclos de subida como los de bajada, ya que no existe diferencias entre ambos. La tabla siguiente (Tabla 4.14) contiene la media de las fuerza leídas para cada masa de carga.

Masa (Kg)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
<Fz> (N)	0	9,88	20	29,84	39,64	49,54	59,32	69,01	78,87	88,86	97,68

Tabla 4.14. Datos promedio de Fz.

A continuación (Gráfica 4.6) representamos los datos de la Tabla 4.14 con el fin de obtener, mediante regresión lineal porque los datos presentan a simple vista de la Gráfica 4.5 una fuerte tendencia lineal, la curva que mejor se aproxima a dichos datos.



Grafica 4.6. Curva calibración de Fz.

Las medidas se ajustan con un coeficiente de correlación lineal del 0,9999 a la recta:

$$\boldsymbol{F_z = 9,8085 \cdot m + 0,288} \quad \text{(ec. 4.5)}$$

Las característica de dicha curva de calibración para Fz, son:

- **Histéresis**: Viendo los datos de la Tabla 4.13 y su respectiva representación de la Gráfica 4.7, el sensor 85M35A3-I40-DH12 de JR3 inc., no presenta histéresis en la medición de la fuerza en el eje "z".

- **Sensibilidad**: $S_{F_z} = \frac{\partial F_z}{\partial m} = 9,8085\ N/kg$

- **Linealidad:**

$$\varepsilon_l = \frac{\Delta F_z}{F_{z\,max}} = \frac{max|F_{z\,real} - F_{z\,teorico}|}{F_{z\,max}}$$

Utilizando ec. 4.5 obtenemos para cada carga su correspondiente valor teórico de Fx, con ellos podemos establecer cuál es el max|Fx real - Fx teórico| (Tabla 4.15).

Masa (Kg)	Fz empírico (N)	Fz teórico (N)	|Fz empírico - Fz teórico| (N)
0	0	0,288	0,288
1	9,88	10,0965	0,2165
2	20,39	19,905	0,485
3	29,84	29,7135	0,1265
4	39,64	39,522	0,118
5	49,54	49,3305	0,2095
6	59,32	59,139	0,181
7	69,01	68,9475	0,0625
8	78,87	78,756	0,114
9	88,86	88,5645	0,2955
10	97,68	98,373	0,693
		max|Fz real – Fz teórico|	**0,693**

Tabla 4.15. Cálculo de error de linealidad para Fz.

$$\varepsilon_{l\,F_z} = \frac{\Delta F_z}{F_{z\,max}} = \frac{max|F_{z\,real} - F_{z\,teorico}|}{F_{z\,max}} = \frac{0,693}{97,68} = 0,00709$$

$$\boldsymbol{\varepsilon_{l\,F_z} = \pm 0,7\%\ FE}$$

- **Exactitud:** De la Tabla 4.16 obtenemos la diferencia máxima entre el valor dado por la curva de calibración y el valor medido experimentalmente. Con dicha diferencia podemos calcular la exactitud del sensor en su medición de Fz.

Masa (kg)	**$F_{teórico}$ con g=9,8**	**$<F_z>$ (N)**	**$\|F_{teórico}-F_z\|$ (N)**
0	0	0	0
1	9,8	9,88	0,08
2	19,6	20,00	0,4
3	29,4	29,84	0,44
4	39,2	39,64	0,44
5	49	49,54	0,54
6	58,8	59,32	0,52
7	68,6	69,01	0,41
8	78,4	78,87	0,47
9	88,2	88,86	0,66
10	98	97,68	0,32
	max(|Fteorico-Fz|)		**0,66**

Tabla 4.16. Cálculo exactitud Fz.

$$\boldsymbol{\varepsilon_{exactitud}} = \frac{\boldsymbol{max(|valor\ medio - valor\ real|)}}{\boldsymbol{valor\ medio\ maximo}}$$

$$\boldsymbol{\varepsilon_{exactitud\ Fx}} = \frac{0{,}66}{97{,}68} = 0{,}006767$$

$$\boldsymbol{\varepsilon_{exactitud\ Fx}} = \mathbf{0{,}7\%}\boldsymbol{FE}$$

- **Zona muerta:** A la vista de la Gráfica 4.6, la curva de calibración para Fz, del mismo modo que ocurría para Fx y Fy, no presenta zonas muertas.

d). Corolario de caracterización de la fuerza en los tres ejes.

En la Tabla 4.17 resumimos las características de Fx, Fy, Fz estudiadas en este ensayo.

	Fx	**Fy**	**Fz**
Curva calibración	**$F_{xsubida}$= 9,3762·m+1,1936 [N]** **$F_{xbajada}$= 9,3938·m+0,5345 [N]**	**$F_{ysubida}$ = 9,2456·m+1,4555[N]** **$F_{ybajada}$ = 9,26·m+0,8945[N]**	**Fz=9,8085·m+0,288 [N]**
Histéresis	**ε_{HFx} = ± 0,6% FS**	**ε_{HFy} = ± 0,6% FS**	**No presenta**
Sensibilidad	**$S_{Fxsubida}$ = 9,3762 N/kg** **$S_{Fxbajada}$ = 9,3938 N/kg**	**$S_{Fysubida}$ = 9,2456N/kg** **$S_{Fybajada}$ = 9,26 N/kg**	**S_{Fz} = 9,8085 N/kg**
Linealidad	**$\varepsilon_{l\ Fx}$ = ± 1,3% FE**	**$\varepsilon_{l\ Fy}$ = ± 1,6 % FE**	**$\varepsilon_{l\ Fz}$= ± 0,7% FE**
Exactitud	**$\varepsilon_{exactiud\ Fx}$ = 4,6%FE**	**$\varepsilon_{exactiud\ Fz}$ = 5,4%FE**	**$\varepsilon_{exactiud\ Fz}$ = 0,7%FE**
Zona muerta	**No presenta**	**No presenta**	**No presenta**
Saturación	**80%FS**	**80%FS**	**80%FS**

Tabla 4.17. Resumen caracterización del sensor 85M35A3-I40-DH12 de JR3 inc. en la medida de fuerza en sus tres ejes.

El error de linealidad dado por el fabricante es de 0,5% sobre fondo de escala. En el eje "z" para la fuerza obtenemos experimentalmente un valor de error de linealidad de 0,7%FS, lo que podemos considerar aceptable con la cantidad de datos obtenidos. No podemos decir lo mismo para Fx y Fy, medidas para las que hemos calculado, respectivamente, un ε_l de 1,3% y 1,6% sobre fondo de escala, lo que supone

alrededor del triple del valor proporcionado en hoja de catálogo. El hecho de que en "z" obtengamos un error de linealidad para la fuerza de acorde con la del fabricante JR3 y, sin embargo, no suceda del mismo modo para Fx y Fy, puede deberse a que en estos últimos existe histéresis, es decir, sus datos se distribuyen de una forma abombada respecto a la medida central, lo que aumenta su no linealidad.

Además, otro motivo causante de esta gran difrencia del error de linealidad en Fx y Fy con respecto al ideal, puede ser la saturación apreciada en la medida de fuerzas en "x" e "y", fenómeno ausente en este ensayo en Fz. Si volvemos a observar los datos promedios (Tabla 4.18):

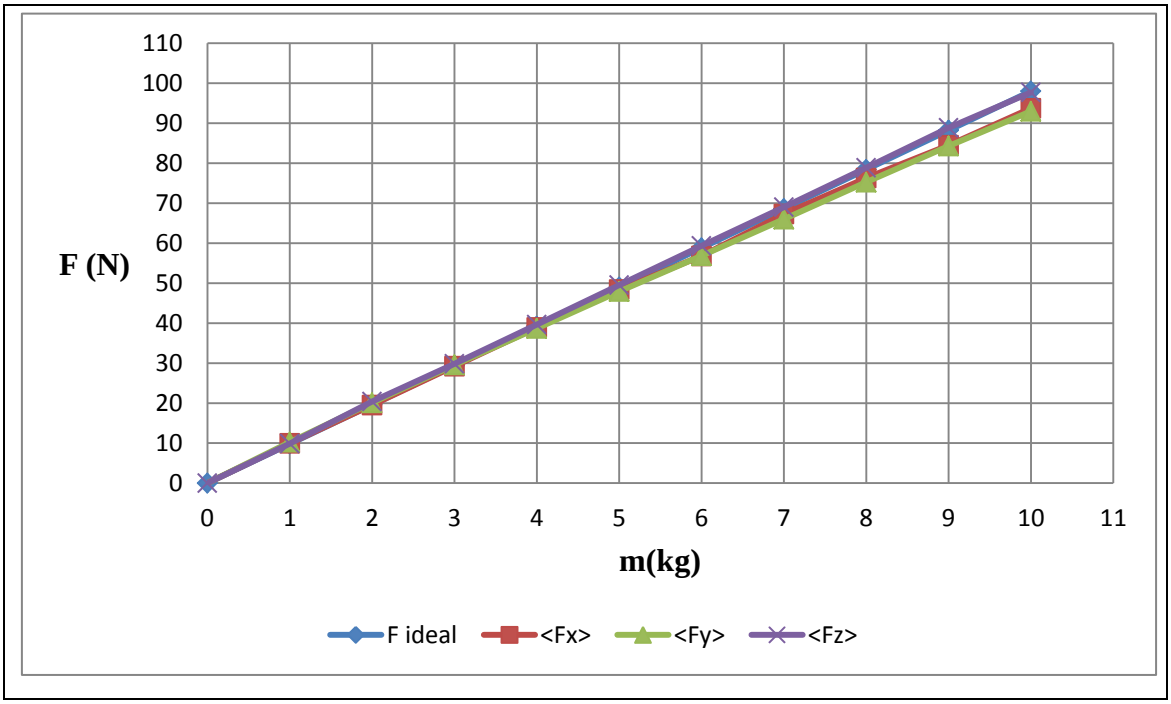
Masa (kg)	**$F_{teorica}$ con g=9,8 m/sg^2**	**<$F_{x\ subida}$> (N)**	**<$F_{x\ bajada}$> (N)**	**<$F_{y\ subida}$> (N)**	**<$F_{y\ bajada}$> (N)**	**<F_z> (N)**
0	0	0	-0,04	0	-0,02	0
1	9,8	10,1	9,78	10,46	9,98	9,88
2	19,6	19,86	19,22	20,3	19,86	20,39
3	29,4	29,68	28,82	29,84	29,28	29,84
4	39,2	39,44	38,38	39,2	38,16	39,64
5	49	48,96	47,82	48,46	47,4	49,54
6	58,8	56,26	57,38	57,12	56,56	59,32
7	68,6	67,68	66,92	66,28	65,78	69,01
8	78,4	76,54	76,06	75,44	75,08	78,87
9	88,2	84,54	84,48	84,46	84,12	88,86
10	98	93,72	93,72	92,96	92,96	97,68

Tabla 4.18. Comparación Fx, Fy, Fz con fuerza teórica para g=9,8 m/sg^2.

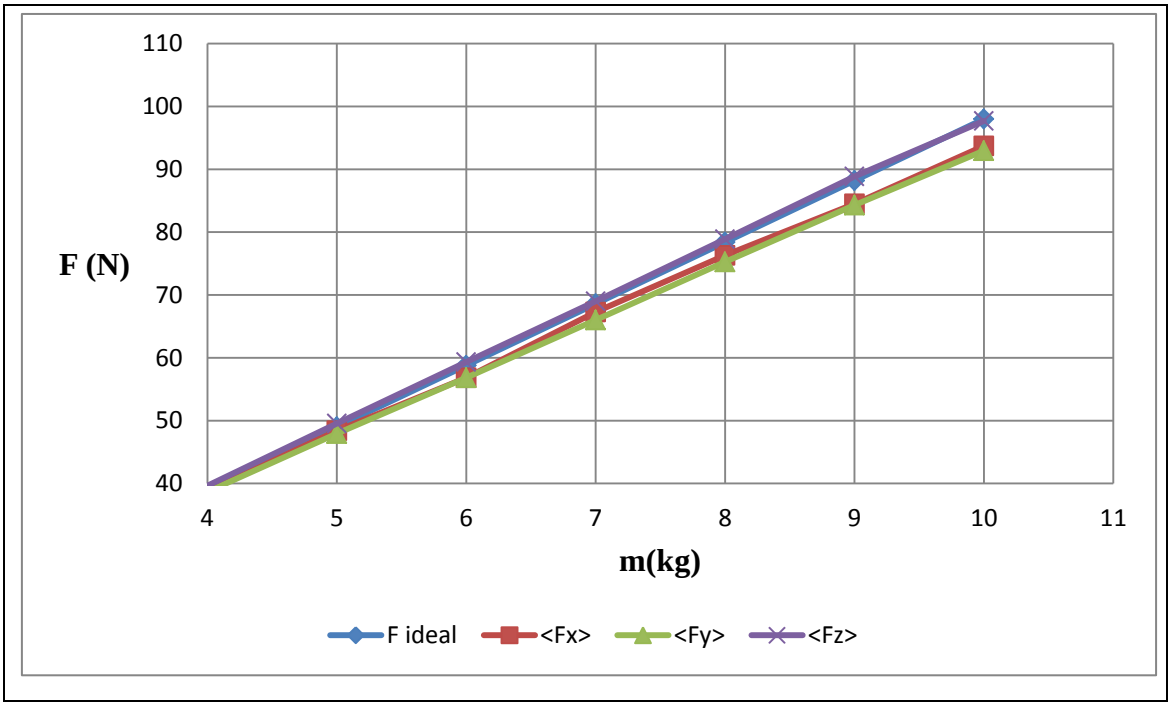
Vemos en la Tabla 4.18 como para el caso de Fx y Fy, no ocurriendo para la fuerza en "z", el sensor va dando medidas cada vez más por debajo del valor ideal correspondiente, lo cual se debe, según se especifica en el manual de la PCI de JR3 inc. (Anexo 2), a que el sensor empieza a saturarse al 80% del fondo de escala. Fx y Fy

tienen un fondo de escala de 63N, se observa en la Gráfica 4.9 y Gráfica 4.10 como los valores leídos en fuerza para "x" e "y" empiezan a saturar en torno a los 50 N (80% de 63 N), haciéndose está saturación más intensa a medida que se aumenta el valor de fuerza. En consecuencia, a partir de este 80%FS los datos tomados para Fx y Fy van semejándose a una suave parábola aumentando la no linealidad del conjunto de datos tomados. En cambio, esta saturación no la apreciamos para Fz, ya que el sensor al poseer en la fuerza medida en "z" un fondo de escala de 126 N su saturación no comienza hasta alrededor de los 100 N, valor que no alcanzamos en nuestro ensayo pues la máxima fuerza máxima ejercida en "z" es de 98 N, es decir, el sensor no entra en la zona de saturación para Fz. Así pues, debido a que Fz no presenta saturación, su error de linealidad no se ve aumentado por este efecto como en los casos de Fx y Fy, de ahí que su no linealidad se ajuste aceptablemente a la dado por el fabricante.

Otra consecuencia negativa de esta saturación, pensamos que puede ser el mayor error de exactitud calculado para el sensor en las medidas de Fx y Fy (4,6% FS y 5,4% FS (Tabla 4.17) respectivamente, 6 y 7 veces mayor con respecto al 0,7%FS correspondiente al calculado para Fz, medidas que no presentan saturación. El error de exactitud se define como la máxima distancia entre los valores leídos con respecto al valor ideal correspondiente que debería proporcionarnos el sensor. Si observamos la Gráfica 4.7 y Gráfica 4.8, en las que se representa las curvas experimentales obtenidas parad Fx, Fy y Fz junto con la curva ideal, vemos como la exactitud del sensor en Fx y Fy va disminuyendo por defecto a medida que aumenta la saturación, hasta tomar un valor de 93,72 N, unos 4 N por debajo del valor ideal de 98 N para una masa de 10 kg. Cosa que no curre para Fz que a lo largo de todo el rango del ensayo solapa a la curva ideal.



Gráfica 4.7. Comparación Fx, Fy, Fz con fuerza teórica para g=9,8 m/sg^2.



Gráfica 4.8. Zoom comparación Fx, Fy, Fz con fuerza teórica para g=9,8 m/sg^2.

4.3.2. Caracterización en momentos.

Para el caso de los momentos Mx, My y Mz, debido a la limitación de medios,únicamente podremos corroborrar que efectivamente el sensor mide bien los momentos alrededor del eje "x" y el "y".

- ¿Por qué no podemos comprobar la medición del momento alrededor del eje "z"? Porque para ello necesitamos un utensilio que permita aplicar fuerzas a una distancia en "x" o en "y" respecto al punto deseado. Expliquémolos más claramente: El momento de una fuerza respecto a un punto es el producto vectorial de dicha fuerza por la distancia a dicho punto, es decir, si tenemos la fuerza F de la Figura 4.20 y queremos calcular su momento respecto al punto O (punto de referencia del sensor) conseguirmos calcular su momento en "y":

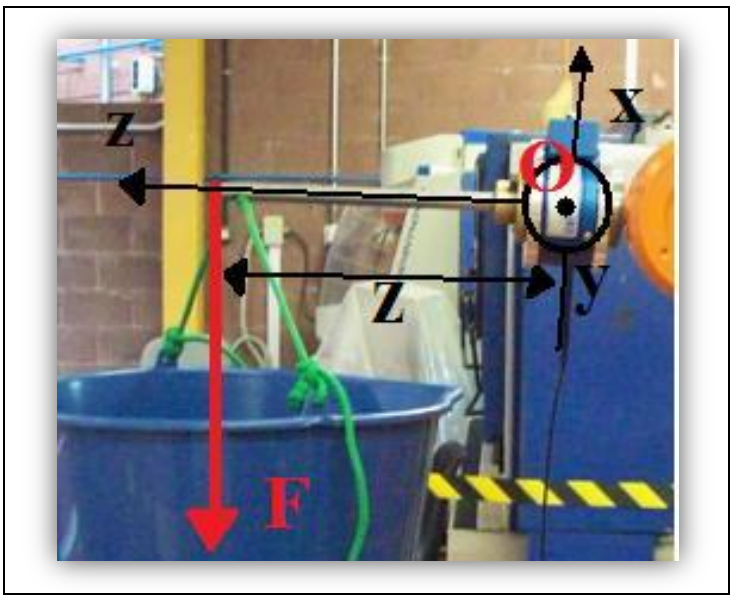


Figura 4.20. Ejemplo cálculo de momentos.

$$\vec{M} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ x & y & z \\ F_x & F_y & F_z \end{vmatrix} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ 0 & 0 & z \\ F_x & 0 & 0 \end{vmatrix} = F_x \cdot z\,\vec{\jmath} \Rightarrow \boldsymbol{\vec{M} = M_y = Fx \cdot z\,\vec{\jmath}}$$

Esta orientación del sensor nos permite calcular el momento alrededor del eje "y", si ahora rotamos el brazo del ABB hasta situar al eje "y" del sensor en la posición vertical podriamos calcular M_x, ya que en esta situacion:

$$\vec{M} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ x & y & z \\ F_x & F_y & F_z \end{vmatrix} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ 0 & 0 & z \\ 0 & F_y & 0 \end{vmatrix} = -F_y \cdot z\,\vec{\imath} \Rightarrow \boldsymbol{\vec{M} = M_x = -Fy \cdot z\,\vec{\imath}}$$

Pero con los medios que disponemos nunca podremos medir momentos alrededor de "z" ya que el utensilio del que se cuelgan las pesas está en la dirección del eje "z", tendríamos que tener un utensilio orientado a lo largo del eje "x" o "y" para conseguir producir momentos alrededor del eje "z".

- ¿Por qué no podemos caracterizas al sensor en su medición de momentos? Porque el sensor, según plano de fabricación (Anexo 5) tiene un fondo de escala para los momentos de 4Nm:

$$FS_{momentos} = FS_{Fx,\,Fy} \cdot 2{,}6in = FS_{Fx,\,Fy} \cdot 2{,}6 \cdot 0{,}0254\ m$$

Al utilizar para nuestro ensayo un fondo de escala para Fx de 50 N y de 55 N para Fy, tenemos que los fondos de escala para los momentos son:

Mx = 3,3 N·m (finalmente seleccionamos un Mx=3 N·m).

My = 3,5 N·m

Mz = 3,5 N·m

Fondos de escalas que nos permiten medir hasta 6,6 N·m para Mx, y 7 N·m para My y Mz. Las fuerzas se aplican en el extremo del utensilio utilizado, que mide 20cm (Figura 4.21), y el sensor toma como referencia su centro geométrico, como según plano de fabricación su espesor es de 35mm, tenemos un total de

0,2m + 0,035/2 m= 0,2175 m de distancia desde el punto de aplicación de la fuerza hasta el punto de referencia que toma el sensor para medir los momentos.

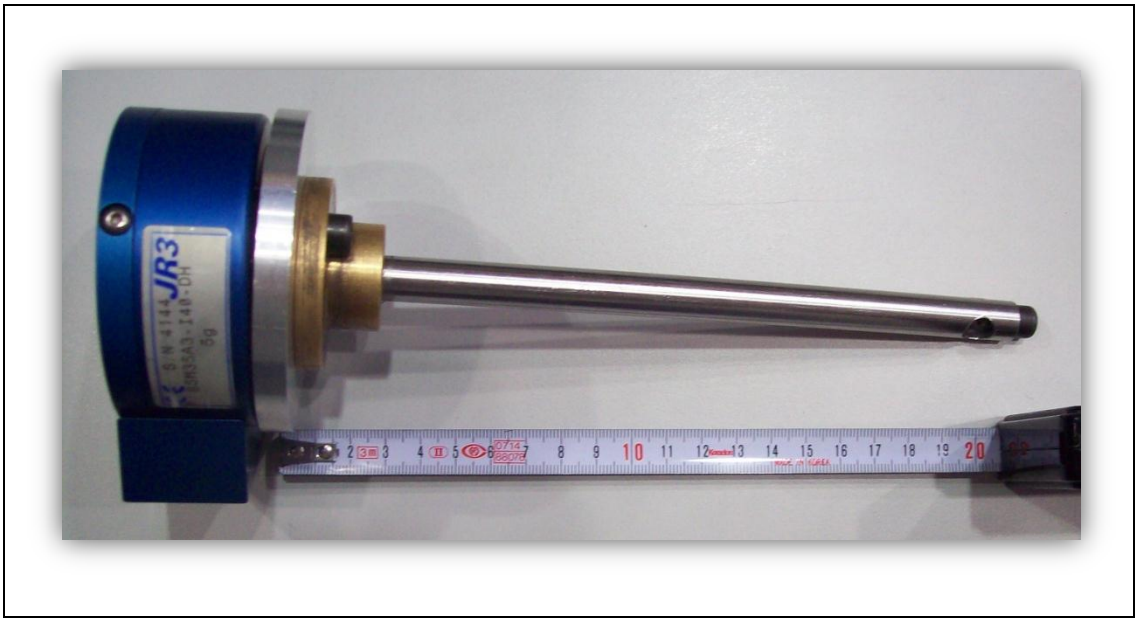


Figura 4.21. Longitud gancho utilizado para colgar los pesos para caracterización.

En consecuencia los rangos máximos se alcanza para:

$$M_{x\,max} = -F_{y\,max} \cdot z\,\vec{\imath} = 6{,}6\ N \cdot m \Rightarrow F_{ymax} = -\frac{6{,}6N \cdot m}{0{,}2175m} = 30{,}34N \Rightarrow$$

$$\Rightarrow \boldsymbol{m = \frac{30,34N}{9,8\ m/sg^2} = 3kg}$$

$$M_{y\,max} = -F_{x\,max} \cdot z\,\vec{\jmath} = 7\ N \cdot m \Rightarrow F_{xmax} = -\frac{7\ N \cdot m}{0{,}2175m} = 32{,}18N \Rightarrow$$

$$\Rightarrow \boldsymbol{m = \frac{32,18N}{9,8\ m/sg^2} = 3,2kg}$$

Únicamente disponemos de pesas de 1kg y 0,5kg, si utilizamos las de 1kg podremos hacer ciclos de seis medidas (tres de subida y tres de bajada) y si utilizamos las de 0,5kg podremos hacer ciclos de 8 medidas (cuatro de subida y cuatro de bajada). Como es obvio, utilizaremos las pesas de 0,5 kg que nos permiten tomar más datos. Sin enmbargo, aún usando las pesas de 0,5 kg, ciclos de tanto sólo 8 medidas no nos permiten llegar a conclusiones medianamente fehacientes, por ello declinamos caracterizar al sensor en su medición de momentos pues nuestra caracterización tendría muy poca sustentación empírica.

Así pues, ante lo explicado, nos limitamos a continuación a demostar que el sensor en estudio mide correctamente los momentos alrededor del eje "x" y del eje "y" (Mx y My).

a). Medidas Mx:

Como ya hemos dicho, para producir momentos alrededor de "x" necesitamos aplicar una fuerza Fy, para ello situamos al sensor en la disposicón de la Figura 4.18. En la Tabla 4.19 se muestran los datos obtenidos para Fy y en la Tabla 4.20 los valores para Mx:

	Fy (N)									
Masa	**1er ciclo**		**2º ciclo**		**3er ciclo**		**4º ciclo**		**5º ciclo**	
(Kg)	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**
0	0	0	0	0	0	0	0	-0,1	0	-0,1
0,5	5,3	5,3	5,3	5,3	5,4	5,4	5,3	5,3	5,3	5,3
1	10,3	10,3	10,3	10,2	10,3	10,2	10,2	10,2	10,3	10,3
1,5	15	15,1	15,1	15	15,1	15,2	15	15,1	15	15,1
2	20	20	20,1	20,1	20,1	20,1	20	20	20	20

Tabla 4.19. Datos Fy para cálculo de Mx.

	Mx (N·m)									
Masa	**1er ciclo**		**2º ciclo**		**3er ciclo**		**4º ciclo**		**5º ciclo**	
(Kg)	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**
0	0	0	0	0	0	0	0	0	0	0
0,5	-1,1	-1,1	-1,1	-1,1	-1,1	-1,1	-1,1	-1,1	-1,1	-1,1
1	-2,2	-2,2	-2,2	-2,2	-2,2	-2,2	-2,2	-2,2	-2,2	-2,2
1,5	-3,2	-3,3	-3,3	-3,2	-3,2	-3,2	-3,2	-3,2	-3,2	-3,2
2	-4,3	-4,3	-4,3	-4,3	-4,3	-4,3	-4,3	-4,3	-4,3	-4,3

Tabla 4.20. Datos Mx.

A simple vista de la Tabla 4.19, se ve como ahora no apreciamos histéresis, lo cual es lógico porque con un fondo de escala de 20N tenemos que tener un error de histéresis de:

$$\varepsilon_{HFy} = \pm 0{,}6\%\ FS = \pm 0{,}6\%\ 20\ N = \pm\ 0{,}12\text{N}$$

Por tanto, ante un rango de medidas tan corto la histéresis no toma protagonismo. Así, para este rango podemos tratar a todos los datos dentro de un mismo ciclo. En la Tabla 4.21 se muestran los valores promedios de Fy y de Mx por masa, así como, los Mx teóricos que deberían provocar las Fy medidas según la ecuación:

$$\vec{M} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ x & y & z \\ F_x & F_y & F_z \end{vmatrix} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ 0 & 0 & z \\ 0 & F_y & 0 \end{vmatrix} = -F_y \cdot z\,\vec{\imath} \Rightarrow \vec{\boldsymbol{M}} = \boldsymbol{M_x} = -\boldsymbol{F_y \cdot z\,\vec{\imath}}\,,$$

con $\mathbf{z = 0{,}2175\ m}$

Masa (Kg)	<Fy> (N)	Mx teorico (N·m)	<Mx empírico> (N·m)
0	-0,02	0,00	0,0
0,5	5,32	-1,16	-1,1
1	10,26	-2,23	-2,2
1,5	15,07	-3,28	-3,2
2	20,04	-4,36	-4,3

Tabla 4.21. Comparativa Mx empírico con Mx teórico.

ºLos datos de las Tabla 4.20 demuestran que el sensor 85M35A3-I40-DH12 de JR3 inc. mide correctamente los momentos alrededor del eje "x" (Mx).

b). Medidas My:

Para la obención de momentos alrededor de "y", situamos al sensor con su eje "x" en vertical (Figura 4.17). Con esta orientación los valores de Fx y My obtenidos son (Tabla 4.22 y Tabla 4.23, respectivamente).

	Fx (N)									
Masa (Kg)	1er ciclo		2º ciclo		3er ciclo		4º ciclo		5º ciclo	
	Subida	Bajada	Subida	Bajada	Subida	Bajada	Subida	Bajada	Subida	Bajada
0	0	0	0	0	0	0	0	0	0	0
0,5	5	5,1	5,1	5,1	5,1	5,1	5	5	5,3	5,3
1	10,3	10,3	10,3	10,3	10,3	10,3	10,3	10,3	10,3	10,3
1,5	15	15,1	15	15	15,1	15,2	15	15,1	15	15,1
2	20	20	20	20	20,1	20,1	20	20	20	20

Tabla 4.22. Datos Fx para cálculo My.

	My (N·m)									
Masa (Kg)	**1er ciclo**		**2º ciclo**		**3er ciclo**		**4º ciclo**		**5º ciclo**	
	Subida	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**	**Subida**	**Bajada**
0	0	0	0	0	0	0	0	0	0	0
0,5	1	1	1	1	1	1	1	1	1	1
1	2,2	2,2	2,2	2,2	2,2	2,2	2,2	2,2	2,2	2,2
1,5	3,2	3,2	3,2	3,2	3,2	3,2	3,2	3,2	3,2	3,2
2	4,3	4,3	4,3	4,3	4,3	4,3	4,3	4,3	4,3	4,3

Tabla 4.23. Datos My.

En la Tabla 4.25 se ve como para este rango tan corto de medidas, Fx no presenta histéresis.

En la siguiente tabla (Tabla 4.24) estudiamos conjuntamente los valores promedios de My con los momentos en "y" teóricos que deberían provocar Fx de acuerdo con la expresión teórica siguiente

$$\vec{M} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ x & y & z \\ F_x & F_y & F_z \end{vmatrix} = \begin{vmatrix} \vec{\imath} & \vec{\jmath} & \vec{k} \\ 0 & 0 & z \\ F_x & 0 & 0 \end{vmatrix} = F_x \cdot z\,\vec{\jmath} \Rightarrow \boldsymbol{\vec{M} = M_y = F_x \cdot z\,\vec{\jmath}},$$

con $\mathbf{z = 0{,}2175\ m}$

Masa (Kg)	**<Fy> (N)**	**Mx teorico (N·m)**	**<Mx empírico> (N·m)**
0	0	0,00	0,0
0,5	5,11	1,11	1,0
1	10,3	2,24	2,2
1,5	15,06	3,28	3,2
2	20,02	4,35	4,3

Tabla 4.24. Comparativa My empírico con My teórico.

De los resultados mostrados en la Tabla 4.23 podemos llegar a la conclusión que el sensor 85M35A3-I40-DH12 de JR3 inc. mide correctamente los momentos alrededor del eje “y”.

4.4. Ensayo en el tobillo.

Con el fin de mostrar las gráficas que nos permite representar la interfaz diseñada, vamos a realizar medidas en la plataforma del tobillo. En posición vertical (Figura 4.22) cargaremos hasta tres kilogramos de uno en un kilogramo. En este ensayo hemos utilizado unos fondos de escala de: FSfx = 60 N, FSfy=52 N, FSfz120 N, FSMx40 Nm, FSMy=37 Nm, FS, Mz=40 Nm.

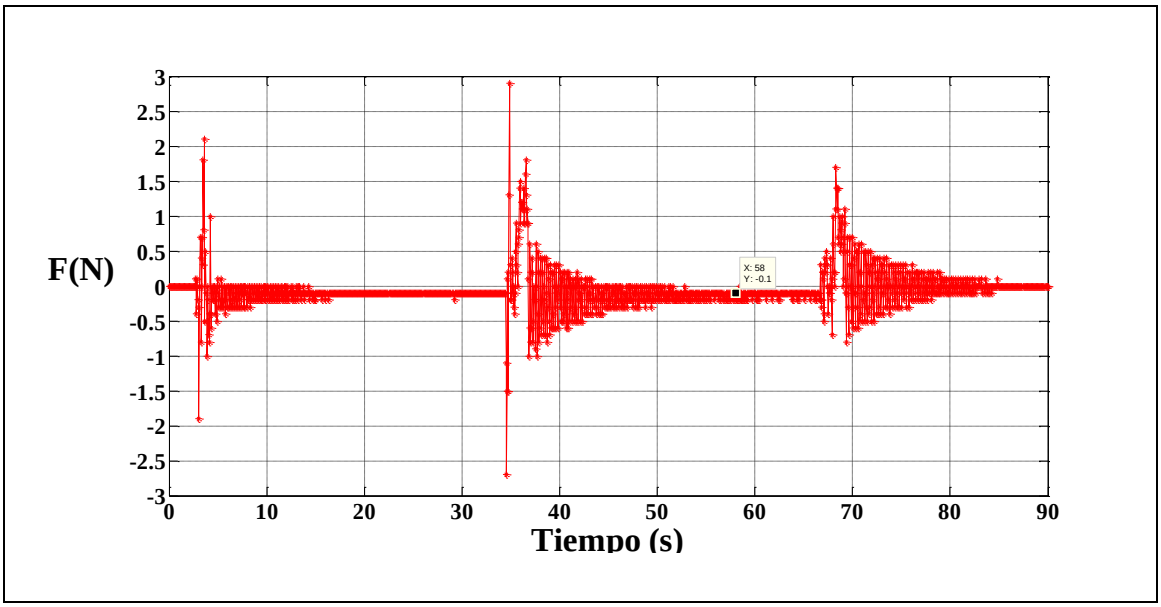
Los datos obtenidos figuran en la Tabla 4.25.

Figura 4.22. Posición tobillo para mostrar gráficas de “interfazJR2_display”.

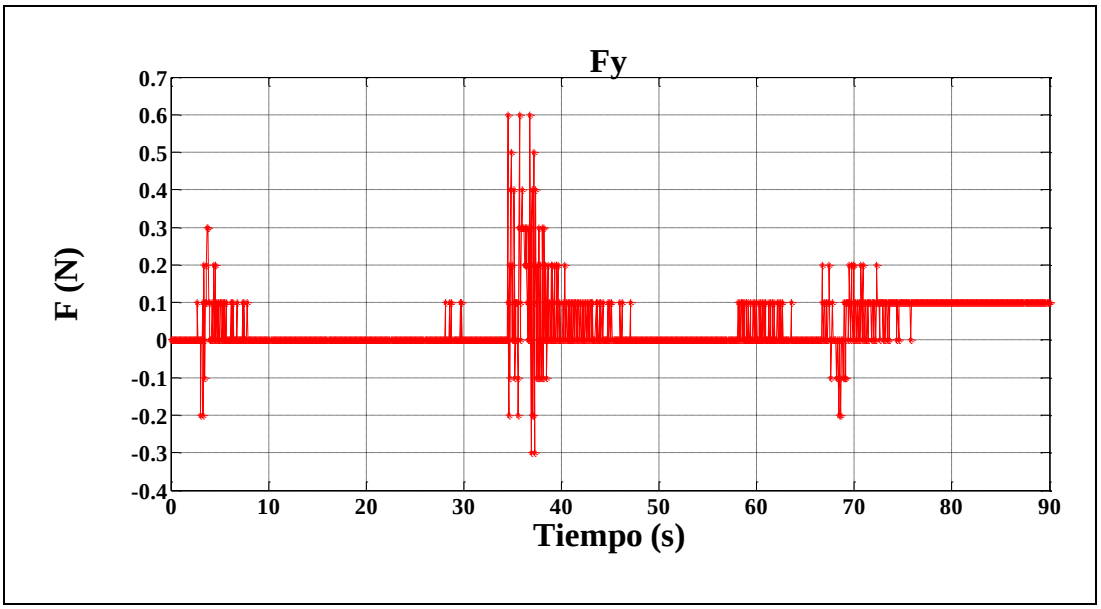
m (kg)	Fx (N)	Fy (N)	Fz (N)	Mx (Nm)	My (Nm)	Mz (Nm)
0	0	0	0	0	0	0
1	-0,1	0	-9,9	0	0	0
2	-0,2	0	-19,9	0	0	0
3	-0,1	0,1	-29,5	0	0	0

Tabla 4.25. Datos ensayo en tobillo.

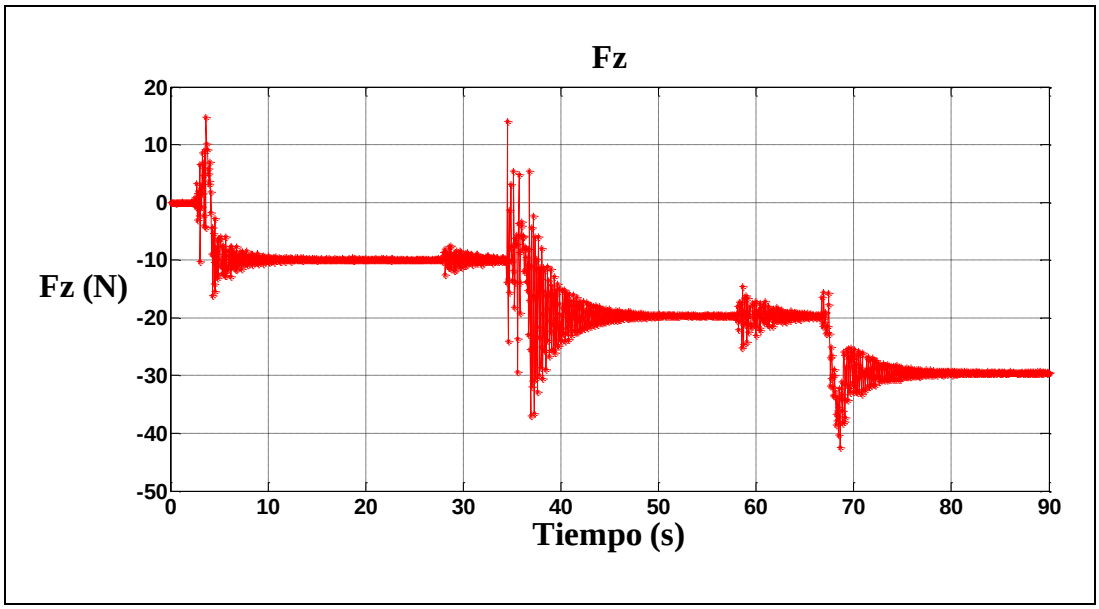
En las siguientes gráficas mostramos la representación respecto al tiempo las distintas fuerzas y momentos.



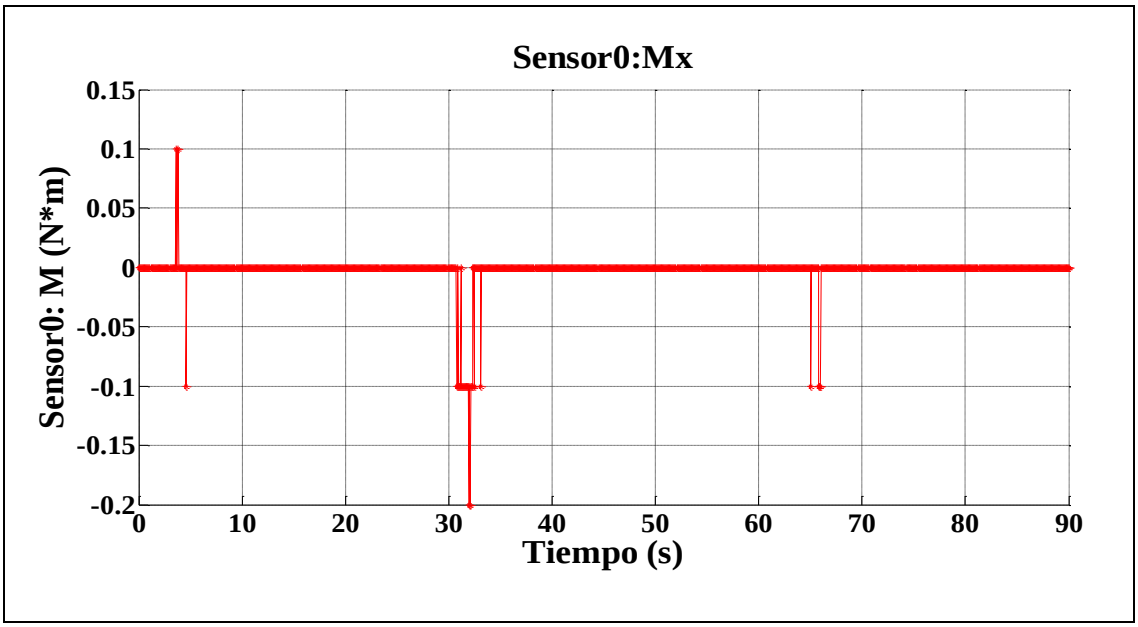
Gráfica 4.9. Fx Vs t.



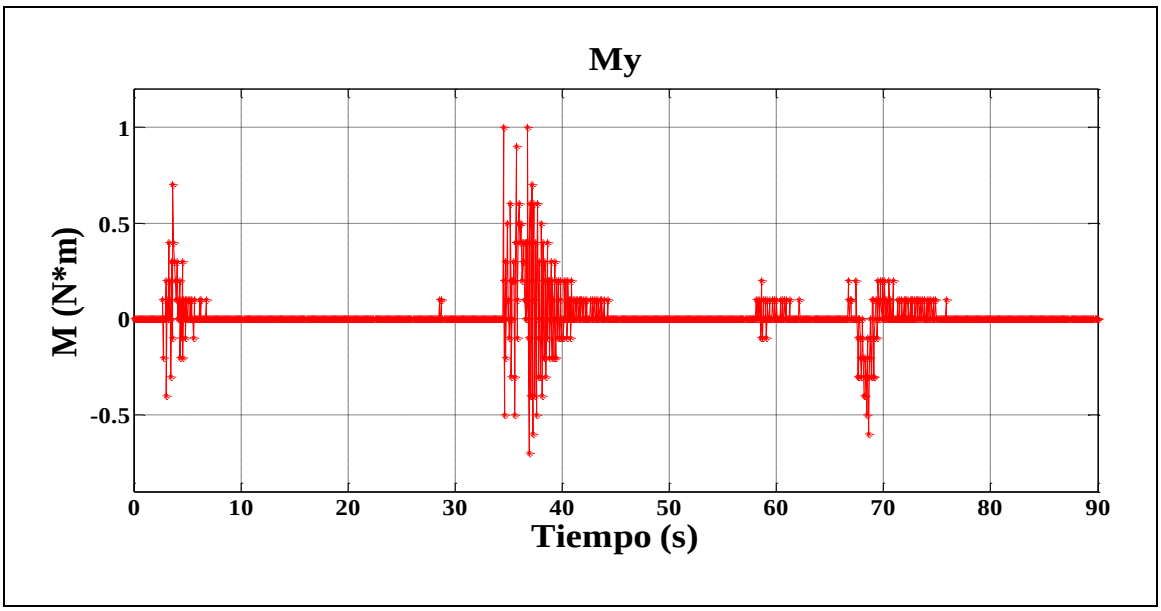
Gráfica 4.10. Fy Vs t.



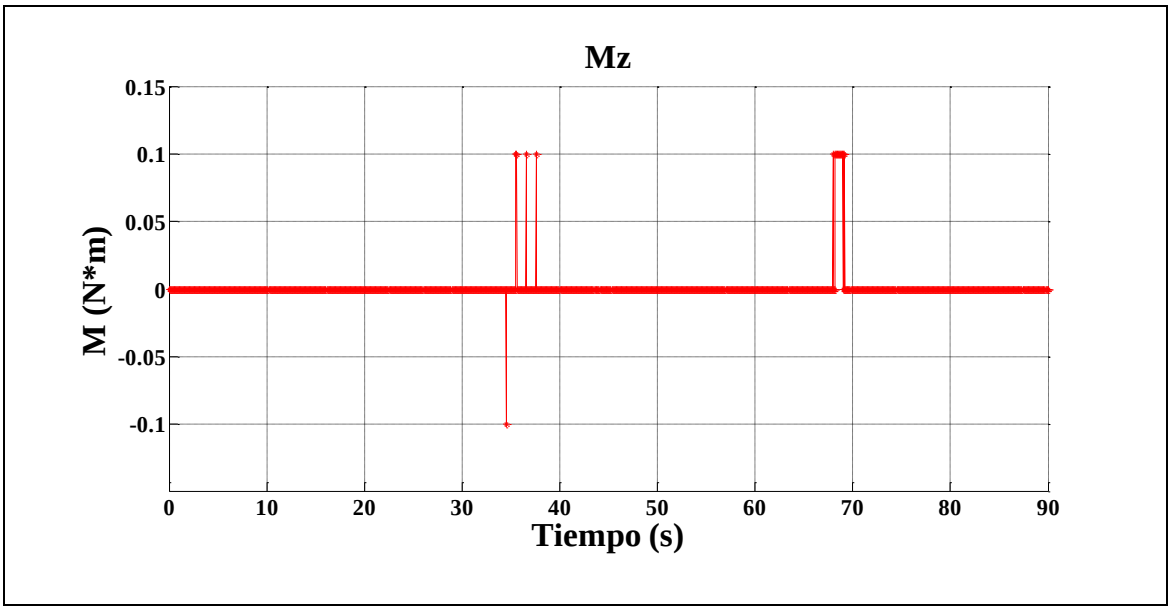
Gráfica 4.11. Fz Vs t.



Gráfica 4.12. Mx Vs t.



Gráfica 4.13. My Vs t.



Gráfica 4.14. Mz Vs t.

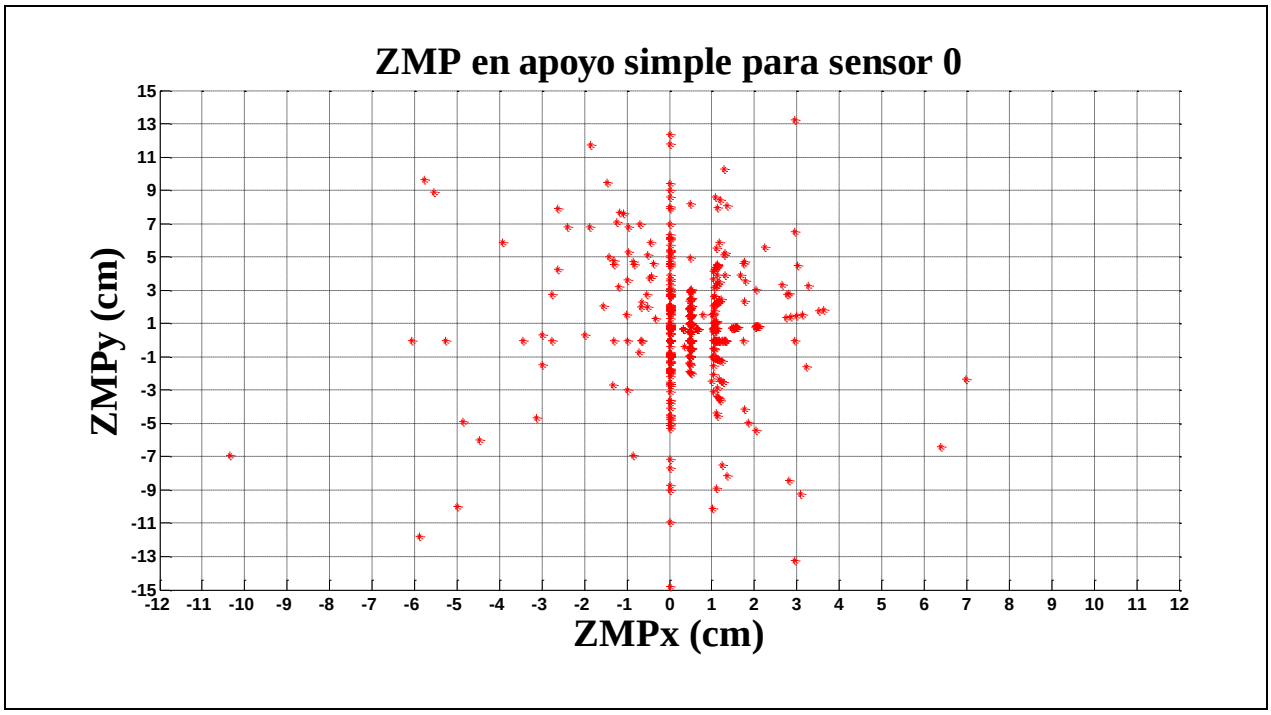
En las gráficas anteriores se aprecia un tiempo de estabilización de las señales de unos 15 sg. En esta posición totalmente en vertical, toda la fuerza, como es lógico, se proyecta en el eje "z". Fx y Fy a lo largo de todo este ensayo se mantienen alrededor de los 0 N (Gráfica 4.9 y Gráfica 4.10), a excepción de cuando se cargan las masas, momentos en los que el ligero tambaleo de la plataforma provoca variaciones de hasta unos ±0.5 N en Fx y Fy, que se estabiliza, poco a poco, de nuevo en 0 N tras unos 15-20 sg. Como consecuencia de la ausencia de fuerza en los ejes "x" e y" tampoco se produce Mz (Gráfica 4.14), de ahí que su señal se mantenga correctamente en 0 Nm. Por su parte, tampoco se producen pares en "x" e "y" (Gráfica 4.12 y Gráfica 4.13) porque la única fuerza que podría provocarlas, Fz, se encuentra en la misma recta que el origen de referencia del sensor, por lo que la distancia entre el punto de aplicación de Fz y los ejes "x" e "y" es nula y, en consecuencia, no hay momentos en estos ejes. Los valores de hasta 1N apreciados en las Gráfica 4.12 y Gráfica 4.13 para Mx y My son provocados por el ya citado tambaleo de la plataforma al cargar los pesos, dichos valores desparecen con al estabilización de la señal.

Para el cálculo del ZMP, como ya se explicó en su momento, utilizamos la expresión siguiente:

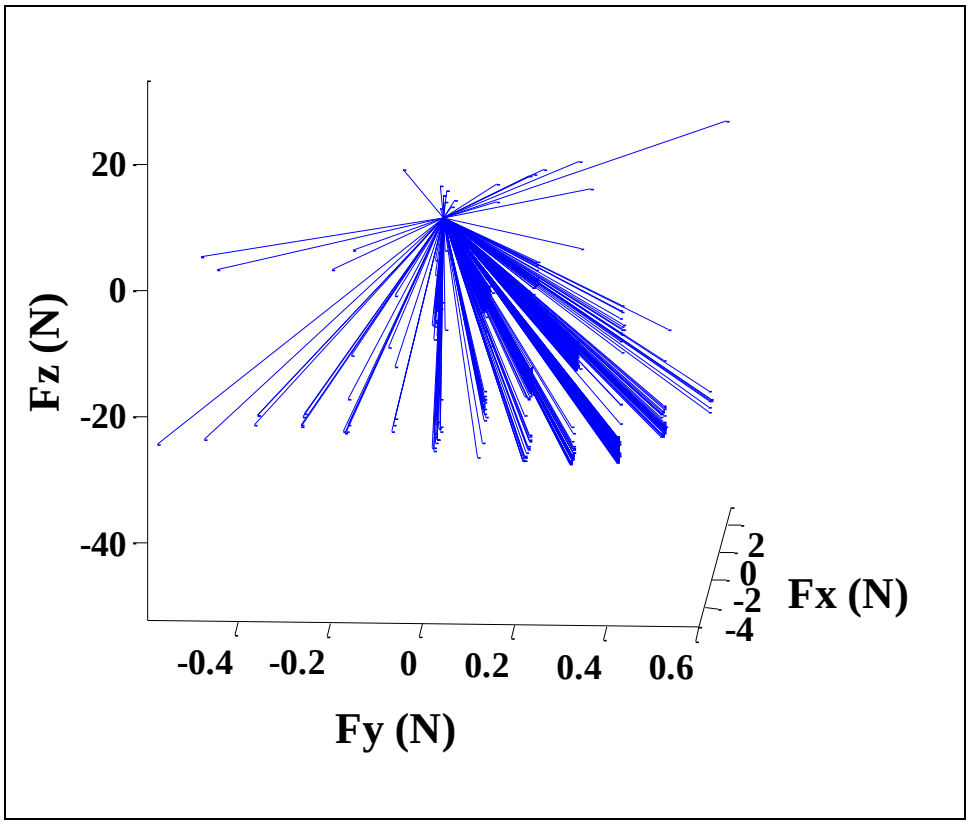
$$\boldsymbol{ZMP} = (\boldsymbol{ZMP_x}, \boldsymbol{ZMP_Y}) = \left(\frac{\boldsymbol{p_z \cdot F_{Ax} - M_{Ay}}}{\boldsymbol{F_{Az}}}, \frac{\boldsymbol{p_z \cdot F_{Ay} + M_{Ax}}}{\boldsymbol{F_{Az}}}\right)$$

En nuestro caso, el sensor está a ras de suelo, así que: pz (altura del tobullo) = 0 mm. Como obtenemos un Mx =My = 0 Nm el ZMP debería encontrase en todo instante en la coordenada (0,0). Sin embargo, en la represemtación instantánea del ZMP de la Gráfica 4.15, vemos que tenemos una gran nube de puntos que oscilan entre los ±1 cm en la coordenada "x" y ±7cm en la "y", muchos de ellos cercanos a (0,0). Además hay ZMP con coordendas muy altas (0,-16), (-10,-7), (-6,-11), (7, -3), (3,11), (3,13), (3,-13), (3,-9) que podrían encontrarse fuera del polígono de soporte (tamaño de la planta de pie para este caso de apoyo simple) lo que povocaría una caída de un robot totalmente en vertical. Esta gran variación del ZMP se debe a los momentos Mx y My que provocamos cuando colocamos la cada carga de 1kg mide y, por tanto, debería no tenerse en cuenta en un estudio de mayor profundidad al considerarlos como valores erróneos provocados por el operario.

Por último, en la Grafica 4.16 se observa todas las resultantes instantáneas de la fuerza, con este gráfico lo único que podemos conseguir es hacernos una idea de cómo ha ido variando cualitativamte la fuerza ejercida.



Gráfica 4.15. ZMP para ensayo tobillo.



Gráfica 4.16. Vector fuerza para ensayo tobillo.

5. Presupuesto.

Como la compañía JR3 inc. es una empresa estadounidense, la compra del dispositivo sensor y de la tarjeta de adquisición de datos se realizará en dólares, por tanto, el coste final en euros dependerá del valor de cambio entre las divisas dólar y euro.

En la Tabla 7.1 se muestra el presupuesto en dólares y en euros utilizado como valor de conversión el correspondiente a 13 de Noviembre de 2011:

1.0 $ = 0,724315 € [17]

	Precio ($)	Precio(€) a 13/11/2011
Sensor fuerza/par 85M35A3-I40-DH12 de JR3 inc.	4485	3248,1
Tarjeta de 4 puertos PCI P/N 1593 de JR3.inc	1680	1216,79
Coste Total	6165	4464,89

Tabla 5.1 Presupuesto del proyecto

6). Conclusiones y trabajos futuros

Nuestro trabajo ha consistido en el diseño y desarrollo para Linux de una aplicación en C++ y Matlab para la adquisición y representación de datos obtenidos por la tarjeta PCI P/N 1593 de JR3 inc. del sensor fuerza/par JR3 inc. que el robot humanoide RH-2 de la Universidad Carlos III de Madrid llevará instalados en tobillos y muñecas. Nosotros en el laboratorio hemos trabajado sobre el modelo 85M35A3-I40-DH12, cuyos rangos de medida son menores al sensor que llevará defnitivamente el robot.

El diseño se estructura en tres módulos:

- Una aplicación C++ (“jr3.cpp”) que se encarga de la adquisición y conversión de datos a la frecuencia de la PCI (20 Hz). Guarda estos valores en dos ficheros, “fichero.txt” y “fichero_Matlab”, el primero de ellos para visionar los datos tras ensayos y el segundo que permite a Matlab guardar todos ellos en una matriz mediante la sentencia en su comando de A=load(“fichero_Matlab.txt”), sabiendo a qué dato corrresponde cada columna el trabajo posterior sobre los mismos se simplifica.

- Por otro lado, el diseño consta de una interfaz gráfica para Matlab (“interfazJR3_display”) con la que el usuario puede interactuar y que recibe de “jr3.cpp” los datos de fuerza/par leídos por el sensor. La interfaz permite trabajar con hasta cuatro sensores a la vez, introducir en tiempo real los fondos de escala deseados y aplicar offset, también visualiza en tiempo real los valores de fuerza y momentos, y representar, posteriormente de haber capturado los valores deseados, las fuerzas y momentos respecto al tiempo. Además, representa el vector fuerza resultante y calcula (en tiempo real) y representa la trayectoria del ZMP en apoyo simple (fase de balanceo) para los dos sensores de los tobillos.

- En medio de las dos aplicaciones anteriores, se encuentra un sistema de comunicación que permite las transferencia de información entre “jr3.cpp” e “interfazJR3_display”. Dicho sistema está formado por una memoria compartida definida en el archivo “memoria.h” donde se guardan lo datos a compartir entre el programa de adquisición de datos y la interfaz. Por un lado, “jr3.cpp” guarda en la memoria los valores de tiempo de toma de muestra, y de las fuerzas y momentos leídos, necesarios todos ellos para que la interfaz los visualice y represente gráficamente. Por otro, “interfaJR3_display” guarda en la memoria, a través de funciones MEX, los valores de fondos de escala, offset y activación de la adquisición de datos, los cuales necesita “jr3.cpp” para iniciar la toma de datos, realizar una conversión adecuada de los mismos e inicializarlos a cero cuando el usuario lo solicita vía interfaz.

  Este sistema de comunicación se complementa con cuatro funciones MEX con las que “interfazJR3_display” se comunida con “memoria.h”. A cada puerto de la PCI le corresponde una de estas mexfunction, las cuales son funciones ejecutables programadas en C/C++ que pueden ser cargadas y ejecutadas por Matlab de forma automática.

Una vez implementada la aplicación de adquisición, visualización y representación gráfica, pasamos a demostrar que efectivamente todo este sistema de sensor, PCI y software diseñado, mide correctamente fuerzas y momentos. Utilizamos el ABB que posee la Universidad Carlos III de Madrid y que nos permitió orientar los ejes del sensor a nuestras necesidades. Con las medidas tomadas, realizamos un pequeño estudio de caracterización del sensor en sus tres ejes para las fuerzas, no así para los momentos, pues debido a limitaciones de medios, únicamente pudimos demostrar su correcta medición. Con la caracterización, concluimos que el sensor presenta histéresis en la medición de fuerza en “x” e “y”. Del mismo modo, corroboramos la existencia de saturación a partir del 80% de fondo de escala, como nos indica el manual del fabricante. La existencia de esta saturación obliga a sobredimensionar el diseño con un sensor que tenga un rango de medición en fuerzas lo suficientemente alto como para

asegurar que las fuerzas provocadas por el humanoide nunca entren en zona de saturación, lo que proporcionaría valores por debajo de los reales.

En una primera versión nuestro sistema completo conseguía visualizar en tiempo real los valores de fuerzas, momentos y el ZMP, a la vez que representaba, también en tiempo real, estos datos. Sin embargo, el tiempo de computación de Matlab para realizar la representación de una única gráfica alcanzaba ya los casi 0,1sg para un número considerable de puntos a pintar, mientras que "jr3.cpp" proprociona un dato cada 0,05sg. En consecuencia, con dicho diseño aumentábamos el periodo de muestreo, lo que no es aceptable para el control en tiempo real del humanoide. Se intentó realizar una segunda versión constituida por dos interfaces, una para la visualización y otra para las representaciones gráficas, y lanzarlas a la vez para que trabajasen en paralelo. Sin embargo, Matlab solamente nos dejaba ejecutar una interfaz. En consecuencia, el diseño final elegido muestra en tiempo real y en todo momento los valores de fuerza/pares mientras que capturamos datos, representando éstos a posteriori.

Por tanto, un trabajo futuro a desarrollar sería investigar sobre la ejecución de interfaces en paralelo que permitan visualizar y representar los datos a la vez y en tiempo real sin aumentar el periodo de muestreo de la PCI. Pero hay que tener en cuenta que las representaciones gráficas, a no ser que se disponga de un PC más potente, siempre van a representar, si se quiere que se haga en tiempo real, menor número de muestra que las obtenidas, ya que MATLAB, como se ha comprobado, tarda casi 0,1sg en representar una sola gráfica, superando los 0,05sg de muestreo.

Otra limitación de la interfaz es que únicamente calcula el ZMP en apoyo simple. Por ello, otra mejora sería implementar el cálculo del ZMP para apoyo doble.

Uno de los mayores problemas que nos ha dado el sensor 85M35A3-I40-DH12 de JR3 inc. ha sido su calibración. Teóricamente, según manual de la compañía, para lo correcta conversión de medidas eléctricos a medidas de fuerzas/pares, hay que introducir el fondo de escala del sensor, en este caso 63N para Fx y Fy, 126 N para Fz y 4.4 Nm para los momentos en los tres ejes. Sin embargo, en la práctica, aunque oscilan en torno a esos valores, cada vez que se conecta el sensor hay que introducir un nuevo

valor de fondo de escala diferente al utilizado en el último ensayo y que diferirá, a su vez, también de los ensayos futuros. Esta variación en los fondo de escala para conseguir una correcta conversión, se observa en los ensayos de calibración y en la medidas tomadas en el tobillo, pues mientras que en el primer caso se tuvo que utilizar unas full scales de 50 N, 55 N, 130 N, 33 Nm, 35 Nm y 33 Nm para Fx, Fy, Fz, Mx, My, Mz, respectivamente; en las medidas tomadas en el tobillo se utilizaron unos valores, respectivos, de 60 N, 52 N, 120 N, 40 Nm, 37 Nm y 40 Nm. Esta necesidad de calibrar el sensor cada vez que se activa nos lleva a proponer otra mejora futura del sistema, consistente en desarrollar un autocalibrado para nuestro sensor una vez instalado en el humanoide. Una propuesta que se hace desde aquí, consistiría en calibrar correctamente los sensores y una vez asegurados que miden bien en los tres ejes tanto las fuerzas como los pares, colocarlos en el robot. Posteriormente llevar al humanoide a una posición que definiremos como patrón de calibrado (por ejemplo, una inclinación del 10% hacia la derecha) y, en esta situación, tomar cuantas más medidas mejor para poder concluir de forma fehaciente los valores de fuerzas y momentos que ejerce el robot sobre sus tobillos en la posición patrón. Así, ya tendríamos unos valores patrones para las fuerzas y momentos. Cada vez que se activase el robot, éste iría a la posición patrón, procesaría los datos leídos por los sensores y seleccionaría unos fondos de escala que diesen lugar a las medidas patrones. De esta forma, nos aseguraríamos de que los sensores van a medir correctamente.

En definitiva, se propone tres mejores a este Proyecto Fin de Carrera:

- Estudiar las posibilidades de conseguir representaciones gráficas en tiempo real.
- Cálculo y representación del ZMP en apoyo doble.
- Y diseño de rutina de autocalibrado.

Referencias

[1] La Enciclopedia. (Ed. Salvat Editores SA. 2003)

[2] Funes Romero, Gonzalo: "Diseño e implementación de los sistemas electrónicos del tren inferior del robot humanoide RH-2". Proyecto Fin de Carrera. Universidad Carlos III de Madrid. 2010.

[3] Navarro Criado, Alberto. "Análisis de los accionadores del robot humanoide RH-2". Proyecto Fin de Carrera. Universidad Carlos III de Madrid. 2009.

[4] Larriva Vásquez, José. Velle Guallpa, Oscar. "Estudio diseño y construcción de un robot bípedo experimental: Análisis y control". Proyecto Fin de Carrera. Universidad Politécnica Salesiana. 2006.

[5]ATI Industrial Automation. Catálogo [en línea]. <http://www.ati-ia.com/products/ft/ft_ModelListing.aspx> [Consulta: 10 de Octubre de 2011].

[6] Schunk. Catálogo [en línea] <http://www.schunk-modular-robotics.com/fileadmin/user_upload/service_robotic/products/sensors/FT-Mini_045_EN.pdf> [Consulta: 13 de Octubre de 2011].

[7] JR3 inc.< http://www.jr3.com/Products.html> [Consulta: 13 de Octubre de 2011]

[8] JR3iNc.Catálogo[en línea]. <http://www.jr3.com/documents/datasheets/sensormodels.PDF> [Consulta: 13 de Octubre de 2011]

[9] Wikipedia http://es.wikipedia.org/wiki/Carrera_a_pie [Consulta: 11 de Octubre de 2011].

[10] Khalil, H. K., (1999). Nonlinear systems.Pearson Education: Upper Saddle River, New Jersey.

[11] Monje Micharet, Concepción A. "Control de la caminata del robot RH-1". Universidad Carlos III de Madrid.

[12] Wikipedia <http://en.wikipedia.org/wiki/Zero_Moment_Point> [Consulta: 16 de Octubre de 2011].

[13] Gómez Royuela, Cristina. "Puesta en marcha y documentación del tobillo del robot humanoide RH-2". Proyecto Fin de Carrera. Universidad Carlos III de Madrid. 2010.

[14] <http://www.rastersoft.com/articulo/pserie.html> [Consulta: 5 de Noviembre de 2011].

[15] García de Jalón, Javier. Rodríguez, José Ignacio. Vidal, Jesús: "Aprenda Matlab 7.0 como si estuviera en primero". (Escuela Técnica Superior de Ingenieros Industriales. Universidad Politécnica de Madrid. 2005).

[16] Pérez García, Miguel A. Álvarez Antón, Juan C. Campo Rodríguez, Juan C. Ferrero Martín, Fco. Javier- Grillo Ortega Gustavo J.:"Instrumentación electrónica" (Thomson, 2004).

[17] http://www.xe.com/ucc/convert/?Amount=1&From=USD&To=EUR

Bibliografía.

1º). Libros

- La Enciclopedia. (Ed. Salvat Editores SA. 2003).
- García de Jalón, Javier. Rodríguez, José Ignacio. Vidal, Jesús: "Aprenda Matlab 7.0 como si estuviera en primero". (Escuela Técnica Superior de Ingenieros Industriales. Universidad Politécnica de Madrid. 2005).
- Pérez García, Miguel A. Álvarez Antón, Juan C. Campo Rodríguez, Juan C. Ferrero Martín, Fco. Javier. Grillo Ortega Gustavo J.:"Instrumentación electrónica" (Thomson, 2004).

2º). Páginas o documentos electrónicos en la red.

- http://www.ati-ia.com
- http://www.schunkmodularrobotics.com/fileadmin/user_upload/service_robotic/products/sensors/FT-Mini_045_EN.pdf
- http://www.jr3.com
- http://www.wikipedia.org
- http://www.rastersoft.com
- http://isa.umh.es/asignaturas/sitr/Tema_Signals.pdf.>
- http://www.xe.com

3º). Proyectos Fin de Carrera y trabajos

- Funes Romero, Gonzalo: "Diseño e implementación de los sistemas electrónicos del tren inferior del robot humanoide RH-2". Proyecto Fin de Carrera. Universidad Carlos III de Madrid. 2010.

- Navarro Criado, Alberto. "Análisis de los accionadores del robot humanoide RH-2". Proyecto Fin de Carrera. Universidad Carlos III de Madrid. 2009.

- Larriva Vásquez, José. Velle Guallpa, Oscar. "Estudio diseño y construcción de un robot bípedo experimental: Análisis y control". Proyecto Fin de Carrera. Universidad Politécnica Salesiana. 2006.

- Gómez Royuela, Cristina. "Puesta en marcha y documentación del tobillo del robot humanoide RH-2". Proyecto Fin de Carrera. Universidad Carlos III de Madrid. 2010.

- García Flores, Guillermo. "Sistema para análisis de la marcha humana". Tesis. Benemérita Universidad Autónoma de Puebla. 2004

- Contreras Bravo, Leonardo Emiro; Roa Garzón, Máximo Alejandro. "Modelamiento de la marcha humana por medio de gráficos de unión" Investigación. 2005

- Monje Micharet, Concepción A. "Control de la caminata del robot RH-1". Universidad Carlos III de Madrid.

ANEXOS

Anexo 1:

Códigos de programación de las aplicaciones diseñadas.

“jr3.cpp”

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <sys/ioctl.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "jr3pci-ioctl.h"

#include "memoria.h"

#define MIN_RATE  1
#define MAX_RATE 1000

static int rate=20;
static int period;
static char *filename;
memo *mem;

void show_usage(char *name)
{
        printf("\n%s [OPTION] [FILE]\n", name);
        printf("-d,--debug\n");
        printf("\tenabled debug\n");
        printf("-h,--help\n");
        printf("\tdisplay this message\n");
        printf("-r,--rate\n");
        printf("\tmessage rate Hz (default=20)\n");
        printf("\[FILE]\n");
        printf("\tFile to store the data\n");
}

int process_args(int argc, char* argv[])
{
        int c;
        while(1) {
                static struct option long_options[]={
                        {"help",0,NULL,'h'},
                        {"rate",1,NULL,'r'},
                        {0,0,0,0},
```

```
                };

                int option_index=0;
                c=getopt_long(argc,argv,"hr:",long_options,&option_index);
                if(c==-1)
                        break;

                switch(c) {
                        case 'h':
                                show_usage(argv[0]);
                                exit(0);
                        case 'r':
                                rate=atoi(optarg);
                                break;
                        default:
                                printf("ERROR: Unknown argument: %c\n",c);
                                break;
                }
        }

        if(argc-optind==1) {
                filename=strdup(argv[optind]);
        }

        return(0);
}

/*Función para representar por terminal los valores de fuerza y par*/
void display_it(int card, six_axis_array saa, int fs[6])
{
        printf("%d: %d %d %d %d %d %d\t\t",
                card,
                10*saa.f[0]*fs[0]/16384,
                10*saa.f[1]*fs[1]/16384,
                10*saa.f[2]*fs[2]/16384,
                saa.m[0]*fs[3]/16384,
                saa.m[1]*fs[4]/16384,
                saa.m[2]*fs[5]/16384);
}

/*Función para guardar en Fichero.txt los valores de fuerza y par*/
void write_it(int card, FILE *fichero, six_axis_array saa,int fs[6])
{
        fprintf(fichero,
                "%d: %d %d %d %d %d %d\t\t",
                card,
                10*saa.f[0]*fs[0]/16384,
                10*saa.f[1]*fs[1]/16384,
                10*saa.f[2]*fs[2]/16384,
                saa.m[0]*fs[3]/16384,
```

```
                saa.m[1]*fs[4]/16384,
                saa.m[2]*fs[5]/16384);
        fflush(fichero);/* La llamada a fflush fuerza al programa a vaciar la caché de
ejecución asociados a fichero.*/

}

/*Función para guardar en fichero_Matlab.txt los valores fuerza y par*/
void write_Matlab(FILE *fichero, six_axis_array saa, int fs[6])
{
        fprintf(fichero,
                "%d %d %d %d %d %d",
                10*saa.f[0]*fs[0]/16384,
                10*saa.f[1]*fs[1]/16384,
                10*saa.f[2]*fs[2]/16384,
                saa.m[0]*fs[3]/16384,
                saa.m[1]*fs[4]/16384,
                saa.m[2]*fs[5]/16384);
        fflush(fichero);/* La llamada a fflush fuerza al programa a vaciar la caché de
ejecución asociados a fichero_Matlab.*/
}

int main(int argc, char* argv[])
{
        FILE *datafile=NULL;
        FILE *fichero;
        FILE *fichero_Matlab;
        int fs0[6];/*fondo de escala para sensor0*/
        int fs1[6];/*fondo de escala para sensor1*/
        int fs2[6];/*fondo de escala para sensor2*/
        int fs3[6];/*fondo de escala para sensor3*/
        int ret;/*ret=retorno de los comando enviados a la PCI, fd es un identificador de
la PCI*/
        int fd;/*Primer parámetro de sentencia ioctl ( ) para envío de comandos a la PCI.
Se trata de un identificador de dispositivo*/
        int signum;
        int shmid;/*Identificador del segmento de la zona de memoria obtenida a través
de la función shmget()*/
        float count=0;/*Determina el número de muestra, junto con period calcularemos
el tiempo de recogida de cada dato(muestra)*/
        struct itimerval my_timer;/*conjunto de señales*/

        six_axis_array fm0, fm1, fm2, fm3;/* fm0, fm1,fm2,fm3 son punteros donde la
PCI ha de retornar los valores de los comandos respectivos IOCTL0_JR3_FILTER0,
IOCTL1_JR3_FILTER0, IOCTL2_JR3_FILTER0, IOCTL3_JR3_FILTER0*/

        sigset_t my_signal_set; /*conjunto de señales para confugurar tiempo de
muestreo*/

        process_args(argc, argv);
```

```
        /* Corroborcion frecuencia dentro de rango */
        if(rate<=MIN_RATE || rate>MAX_RATE) {
                printf("Rate %d out of range (%d>rate<=%d)\n",
                        rate, MIN_RATE, MAX_RATE);
                exit(1);
        }

        /* Configuracion de la PCI*/
        if ((fd = open("/dev/jr3",O_RDWR)) < 0) {
                printf("Can't open device. No way to read force!\n");
                exit(1);
        }

        /* Apertura fichero donde guardaremos los datos */
        fichero=fopen("fichero.txt","w");
        if (fichero == NULL)
        {
        printf ("Error. Can't open fichero file.\n");
        }

                fichero_Matlab=fopen("fichero_Matlab.txt","w");
        if (fichero_Matlab == NULL)
        {
        printf ("Error. Can't open fichero file.\n");
        }

        //Creación de la zona de memoria compartida
        if((shmid = shmget(CLAVE_SHM,sizeof(memo),IPC_CREAT|0777)) == -1)
/*proporciona el identificador de segmento de la zona de memoria compartida asociado
a key*/
        {
      perror("shmget");
      exit(EXIT_FAILURE); // Error al crear la memoria compartida
  }
  mem = (memo *)shmat(shmid,0,0); // Obtención del puntero a la estructura de datos
compartida


 /*Inicializamos los valores de las fuerzas y momentos a cero*/
        ret=ioctl(fd,IOCTL0_JR3_ZEROOFFS);/*offset sensor0:Enviamos el comando
IOCTL_JR3_ZEROFFS a la PCI que inicializa a cero los valores de fuerza y momentos
del sensor0*/
        ret=ioctl(fd,IOCTL1_JR3_ZEROOFFS);/*offset sensor1:Enviamos el comando
IOCTL_JR3_ZEROFFS a la PCI que inicializa a cero los valores de fuerza y momentos
del sensor1 */
        ret=ioctl(fd,IOCTL2_JR3_ZEROOFFS);/*offset sensor2:Enviamos el comando
IOCTL_JR3_ZEROFFS a la PCI que inicializa a cero los valores de fuerza y momentos
del sensor2*/
```

```
        ret=ioctl(fd,IOCTL3_JR3_ZEROOFFS);/*offset sensor3:Enviamos el comando IOCTL_JR3_ZEROFFS a la PCI que inicializa a cero los valores de fuerza y momentos del sensor3*/


        /* configuracion de las señales y timers para toma periódica de datos(tiempo muestreo) */
        sigemptyset(&my_signal_set);     /* Inicia un conjunto de señales excluyendo toda señal*/
        sigaddset(&my_signal_set, SIGALRM); /*Añade la señal SIGALRM(señal de fin de temporización)al conjunto de señales previamente iniciado */
        pthread_sigmask(SIG_SETMASK, &my_signal_set, NULL); /*Bloqueo de la señal:my_signal_set*/

        /* Periodo en usg */
        period = 1000000/rate;

        /* Tiempo entre alarmas */
        my_timer.it_interval.tv_sec = 0;
        my_timer.it_interval.tv_usec = period;/*cada vez que llega  a cero el temporizador se reinicia con un valor del period(periodo de muestreo, 0.05sg)*/

/* Inicio primera alarma */
        my_timer.it_value.tv_sec = 0;
        my_timer.it_value.tv_usec = period;

        setitimer(ITIMER_REAL, &my_timer, NULL);/*SIGALARM permite controlar intervalos de tiempo medidos en segundos. Se utiliza setitimer poruqe nos permite controlar intervalos de usg setitimer ITIMER_REAL temporizador que disminuye ne tiempo real, cuando llega a cero produce la señal SIGALRM*/

        printf("Starting with rate=%d Hz, period=%d mseg...\n", rate, 1000/rate);

        /*Cabecera de fichero.txt donde se guardan los datos*/
        fprintf(fichero, "Starting with rate=%d Hz, period=%d mseg...\n\n", rate, 1000/rate);
        fprintf(fichero, " El primer digito indica el tiempo en segundos\n");
        fprintf(fichero, " De cada grupo el primer digito seguido de dos puntos indica el numero de sensor\n");
        fprintf(fichero, " Los 6 últimos digitos de cada sensor son consecutivamente: Fx, Fz, Mx, My, Mz\n\n ");
        fprintf(fichero, " Fuerzas en 10*N\n");
        fprintf(fichero, "  Momentos en 10*N*m\n\n");
        fprintf(fichero, "Tiempo(sg)\n");
   fprintf(fichero,"   sensor0\t\t sensor1\t\t  sensor2\t \t   sensor3\n\n");
   fflush(fichero);

do  {
if(mem->on==1)
{
```

```
                sigwait(&my_signal_set, &signum);

                printf("%f\t",count*period/1000000);/*Imprimimos en terminal el tiempo de recogida de cada dato*/

    fprintf(fichero,"%f\t",count*period/1000000);/*Guardamos en fihcero.txt el tiempo de recogida de cada dato*/
    fflush(fichero);

    fprintf(fichero_Matlab,"%f\t",count*period/1000000);/*Guardamos en fihcero_Matlab.txt el tiempo de recogida de cada dato*/
    fflush(fichero_Matlab);
    mem->tiempo=count*period/1000000;/*Guardamos en la memoria compartida el tiempo de recogida de cada dato*/

    /* Introducción fondo de escalas*/
    /*-----------SENSOR0---------*/
                if (mem->set_fullscales0==1)
                {
                fs0[0]=mem->fsx0;
                fs0[1]=mem->fsy0;
                fs0[2]=mem->fsz0;
                fs0[3]=mem->Msx0;
                fs0[4]=mem->Msy0;
                fs0[5]=mem->Msz0;
                }


        /*-----------SENSOR1---------*/
        if(mem->set_fullscales1==1)
        {
                fs1[0]=mem->fsx1;
                fs1[1]=mem->fsy1;
                fs1[2]=mem->fsz1;
                fs1[3]=mem->Msx1;
                fs1[4]=mem->Msy1;
                fs1[5]=mem->Msz1;
        }

        /*-----------SENSOR2---------*/
        if(mem->set_fullscales2==1)
        {
                fs2[0]=mem->fsx2;
                fs2[1]=mem->fsy2;
                fs2[2]=mem->fsz2;
                fs2[3]=mem->Msx2;
                fs2[4]=mem->Msy2;
                fs2[5]=mem->Msz2;
        }
```

```
          /*-----------SENSOR3---------*/

        if(mem->set_fullscales3==1)
        {
                fs3[0]=mem->fsx3;
                fs3[1]=mem->fsy3;
                fs3[2]=mem->fsz3;
                fs3[3]=mem->Msx3;
                fs3[4]=mem->Msy3;
                fs3[5]=mem->Msz3;
        }
                printf("full_scales0=%d %d %d %d %d %d\n", fs0[0],
fs0[1],fs0[2],fs0[3],fs0[4],fs0[5]);
                printf("full_scales1=%d %d %d %d %d %d\n", fs1[0],
fs1[1],fs1[2],fs1[3],fs1[4],fs1[5]);
                printf("full_scales2=%d %d %d %d %d %d\n", fs2[0],
fs2[1],fs2[2],fs2[3],fs2[4],fs2[5]);
                printf("full_scales3=%d %d %d %d %d %d\n", fs3[0],
fs3[1],fs3[2],fs3[3],fs3[4],fs3[5]);


/*------------------------------OFFSET-------------------------------------------------------*/
/* Se hace offset de fuerza y momento si se ha presionado en Matlab botón
correspondiente*/

/*---------Offset sensor 0---------------*/
                if(mem->offset0==1){
                ret=ioctl(fd,IOCTL0_JR3_ZEROOFFS);
                mem->offset0=0;
                }

/*---------Offset sensor 1---------------*/
if(mem->offset1==1){
                ret=ioctl(fd,IOCTL1_JR3_ZEROOFFS);
                mem->offset1=0;
                }

/*---------Offset sensor 2---------------*/
if(mem->offset2==1){
                ret=ioctl(fd,IOCTL2_JR3_ZEROOFFS);
                mem->offset2=0;
                }

/*---------Offset sensor 3---------------*/
                if(mem->offset3==1){
                ret=ioctl(fd,IOCTL3_JR3_ZEROOFFS);
                mem->offset3=0;
                }
```

```
 /* Guardamos en la memoria compartida las fuerzas y momentos leídos por sensor0*/
/*-----------------------------------------SENSOR0-----------------------------------------
-------------------------*/
                ret=ioctl(fd,IOCTL0_JR3_FILTER0,&fm0);/* La PCI guarda en el
puntero &fm0 el valor del flitro0 del sensor0, cuyo acceso se hace mediante

                        el comando IOCTL0_JR3_FILTER0*/

                mem->Fx0 =10*fm0.f[0]*fs0[0]/16384;  /*Al recibir una medida, hay
que aplicarle un factor de escala para conseguir el valor real. Esta conversión necesita
los full scales. Dicha conversión está indicada en manual JR3.PCI con n=14 bits lo que
supone una resolución de 1/2^n 1/16384*/
                mem->Fy0 =10*fm0.f[1]*fs0[1]/163384;
                mem->Fz0 =10*fm0.f[2]*fs0[2]/16384;

                mem->Mx0 = fm0.m[0]*fs0[3]/16384;
                mem->My0 = fm0.m[1]*fs0[4]/16384;
                mem->Mz0 = fm0.m[2]*fs0[5]/16384;

                 /* Escribimos en terminal los valores del sensor1*/
                display_it(0, fm0,fs0);

                 /*Guardamos los valores del sensro0 en fichero y fichero_Matlab*/
                write_it(0,fichero, fm0, fs0);
                write_Matlab(fichero_Matlab, fm0, fs0);

/*-----------------------------------------SENSOR1-----------------------------------------*/
                ret=ioctl(fd,IOCTL1_JR3_FILTER0,&fm1);/*La PCI guarda en el
puntero &fm1 el valor del flitro0 del sensor01, cuyo acceso se hace mediante el
comando IOCTL1_JR3_FILTER0*/

                mem->Fx1 =10*fm1.f[0]*fs1[0]/16384;
                mem->Fy1 =10*fm1.f[1]*fs1[1]/16384;
                mem->Fz1 =10*fm1.f[2]*fs1[2]/16384;

                mem->Mx1 = fm1.m[0]*fs1[3]/16384;
                mem->My1 = fm1.m[1]*fs1[4]/16384;
                mem->Mz1 = fm1.m[2]*fs1[5]/16384;

        /* Escribimos en terminal los valores del sensor1*/
                display_it(1, fm1,fs1);

        /*Guardamos los valores del sensro1 en fichero y fichero_Matlab*/
          write_it(1, fichero, fm1,fs1);
                write_Matlab(fichero_Matlab, fm1, fs1);
```

```
/*-----------------------------------------SENSOR2-----------------------------------------*/
                ret=ioctl(fd,IOCTL2_JR3_FILTER0,&fm2);/*La PCI guarda en el
puntero &fm3 el valor del flitro0 del sensor3, cuyo acceso se hace mediante el comando
IOCTL3_JR3_FILTER0*/

                mem->Fx2 =10*fm2.f[0]*fs2[0]/16384;
                mem->Fy2 =10*fm2.f[1]*fs2[1]/16384;
                mem->Fz2 =10*fm2.f[2]*fs2[2]/16384;

                mem->Mx2 = fm2.m[0]*fs2[3]/16384;
                mem->My2 = fm2.m[1]*fs2[4]/16384;
                mem->Mz2 = fm2.m[2]*fs2[5]/16384;

        /* Escribimos en terminal los valores del sensor2*/
                display_it(2,fm2,fs2);

        /*Guardamos los valores del sensro2 en fichero y fichero_Matlab*/
                write_it(2,fichero, fm2, fs2);
                write_Matlab(fichero_Matlab, fm2, fs2);

/*-----------------------------------------SENSOR3-----------------------------------------*/
                ret=ioctl(fd,IOCTL3_JR3_FILTER0,&fm3);/*La PCI guarda en el
puntero &fm0 el valor del flitro0 del sensor0, cuyo acceso se hace mediante el comando
IOCTL0_JR3_FILTER0*/

                mem->Fx3 =10*fm3.f[0]*fs3[0]/16384;
                mem->Fy3 =10*fm3.f[1]*fs3[1]/16384;
                mem->Fz3 =10*fm3.f[2]*fs3[2]/16384;

                mem->Mx3 = fm3.m[0]*fs3[3]/16384;
                mem->My3 = fm3.m[1]*fs3[4]/16384;
                mem->Mz3 = fm3.m[2]*fs3[5]/16384;

        /* Escribimos en terminal los valores del sensor3*/
                display_it(3, fm3,fs3);
                write_it(3,fichero, fm3, fs3);
                write_Matlab(fichero_Matlab, fm3, fs3);
/*-------------------------------------------------------------------------------------------*/

                printf("\n\n");
                fprintf(fichero,"\n\n");
                fflush(fichero);
          fprintf(fichero_Matlab,"\n");
          fflush(fichero_Matlab);
                count++;/*aumentamos contador para saber tiempo de recogida de cada
dato*/
        }

        } while(1);
```

```
	if (datafile!=NULL) {
		fclose(datafile);
	}

	if (fichero!=NULL) {
		fclose(fichero);
	}



	printf("Stopping.\n");

	return 1;
}
```

“interfazJR3_display.m”

```
function varargout = interfazJR3_display(varargin)
% INTERFAZJR3_DISPLAY M-file for interfazJR3_display.fig

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @interfazJR3_display_OpeningFcn, ...
                   'gui_OutputFcn',  @interfazJR3_display_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
   gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
   [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
   gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before interfazJR3_display is made visible.
function interfazJR3_display_OpeningFcn(hObject, eventdata, handles, varargin)

%Activación de la tarjeta PCI
! make
! insmod jr3pci.ko
! make node

borrado(); %Elimina la zona de memoria compartida para evitar errores posteriores
%! ./jr3 &
inicializacion();

handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% --- Outputs from this function are returned to the command line.
function varargout = interfazJR3_display_OutputFcn(hObject, eventdata, handles)

varargout{1} = handles.output;

% --- Executes on button press in HELP.
function HELP_Callback(hObject, eventdata, handles)
open('help.txt')
```

```
%--------------------ON/OFF-----------------------------------------------
% --- Executes on button press in ON.
function ON_Callback(hObject, eventdata, handles)
global tiempo

global fsx0 fsy0 fsz0 Msx0 Msy0 Msz0  Fx0 Fy0 Fz0 Mx0 My0 Mz0
global fsx1 fsy1 fsz1 Msx1 Msy1 Msz1  Fx1 Fy1 Fz1 Mx1 My1 Mz1

global fsx2 fsy2 fsz2 Msx2 Msy2 Msz2  Fx2 Fy2 Fz2 Mx2 My2 Mz2
global fsx3 fsy3 fsz3 Msx3 Msy3 Msz3  Fx3 Fy3 Fz3 Mx3 My3 Mz3

global offset0 offset1 offset2 offset3
global set_offset0  set_offset1  set_offset2   set_offset3

global  set_fullscales0 set_fullscales1 set_fullscales2 set_fullscales3

global start01 start23

global  set_pz0 set_pz1 ZMPx0 ZMPy0 ZMPx1 ZMPy1 pz0 pz1

global on

on=1;

while on==1
   tic

   %----------------------------DISPLAY SENSOR0-----------------------------------------
---
   if set_fullscales0==1 %Si se ha pulsado el botÃ³n set-fullsacles0 en la interfaz,
entonces set_fullscales0=1
      if set_offset0==1 %Si se ha pulsado el botÃ³n set_offset0 en la interfaz, entonces
set_offset0=1
         offset0=1;
      else
         offset0=0;
      end

     %Enviamos mediante la funcion MEX leer_datos0: offset,on, set_fullscales y full
scales a memoria compartida
     %Guardamos en tiempo,Fx0,Fy0,Fz0,Mx0,My0,Mz0 los valores de tiempo, fuerzas
y momentos gurados por "jr3.cpp" en memoria compartida

[tiempo,Fx0,Fy0,Fz0,Mx0,My0,Mz0]=leer_datos0(offset0,on,set_fullscales0,fsx0,fsy0,f
sz0,Msx0,Msy0,Msz0);

       Fx0=Fx0/10;
       Fy0=Fy0/10;
       Fz0=Fz0/10;
       Mx0=Mx0/10;
```

```
        My0=My0/10;
        Mz0=Mz0/10;

      %Mostramos en el display correspondiente los valores de fuerzas y momentos
        set(handles.Fx0, 'String',num2str(Fx0));
        set(handles.Fy0, 'String',num2str(Fy0));
        set(handles.Fz0, 'String',num2str(Fz0));

        set(handles.Mx0, 'String',num2str(Mx0));
        set(handles.My0, 'String',num2str(My0));
        set(handles.Mz0, 'String',num2str(Mz0));

        offset0=0;
        set_offset0=0;

    end

    %--------------------------DISPLAY SENSOR1----------------------------
    if set_fullscales1==1
       if set_offset1==1
            offset1=1;
        else
            offset1=0;
        end


[tiempo,Fx1,Fy1,Fz1,Mx1,My1,Mz1]=leer_datos1(offset1,on,set_fullscales1,fsx1,fsy1,f
sz1,Msx1,Msy1,Msz1);

        Fx1=Fx1/10;
        Fy1=Fy1/10;
        Fz1=Fz1/10;
        Mx1=Mx1/10;
        My1=My1/10;
        Mz1=Mz1/10;

      set(handles.Fx1, 'String',num2str(Fx1));
      set(handles.Fy1, 'String',num2str(Fy1));
      set(handles.Fz1, 'String',num2str(Fz1));

      set(handles.Mx1, 'String',num2str(Mx1));
      set(handles.My1, 'String',num2str(My1));
      set(handles.Mz1, 'String',num2str(Mz1));

      offset1=0;
      set_offset1=0;
  end
```

```
%--------------------------DISPLAY SENSOR2-----------------------------

   if set_fullscales2==1

      if set_offset2==1
        offset2=1;
      else
        offset2=0;
      end


[tiempo,Fx2,Fy2,Fz2,Mx2,My2,Mz2]=leer_datos2(offset2,on,set_fullscales2,fsx2,fsy2,f
sz2,Msx2,Msy2,Msz2);

        Fx2=Fx2/10;
        Fy2=Fy2/10;
        Fz2=Fz2/10;
        Mx2=Mx2/10;
        My2=My2/10;
        Mz2=Mz2/10;

      set(handles.Fx2, 'String',num2str(Fx2));
      set(handles.Fy2, 'String',num2str(Fy2));
      set(handles.Fz2, 'String',num2str(Fz2));

      set(handles.Mx2, 'String',num2str(Mx2));
      set(handles.My2, 'String',num2str(My2));
      set(handles.Mz2, 'String',num2str(Mz2));

      offset2=0;
      set_offset2=0;
   end
    %--------------------------DISPLAY SENSOR3-----------------------------
    if set_fullscales3==1

      if set_offset3==1
        offset3=1;
      else
        offset3=0;
      end


[tiempo,Fx3,Fy3,Fz3,Mx3,My3,Mz3]=leer_datos3(offset3,on,set_fullscales3,fsx3,fsy3,f
sz3,Msx3,Msy3,Msz3);

        Fx3=Fx3/10;
        Fy3=Fy3/10;
        Fz3=Fz3/10;
        Mx3=Mx3/10;
        My3=My3/10;
```

```
        Mz3=Mz3/10;

       set(handles.Fx3, 'String',num2str(Fx3));
       set(handles.Fy3, 'String',num2str(Fy3));
       set(handles.Fz3, 'String',num2str(Fz3));

       set(handles.Mx3, 'String',num2str(Mx3));
       set(handles.My3, 'String',num2str(My3));
       set(handles.Mz3, 'String',num2str(Mz3));

       offset3=0;
       set_offset3=0;
    end

    %-----------------------CALCULO ZMP----------------------------------
   if set_pz0==1 %Â¿Se ha introducido valor en interfaz para calcular ZMP del sensor0?
      ZMPx0=100*((pz0*Fx0)/1000 - My0)/Fz0;
      ZMPy0=100*((pz0*Fy0)/1000 + Mx0)/Fz0;
      set(handles.ZMPx0,'String',num2str(ZMPx0));
      set(handles.ZMPy0,'String',num2str(ZMPy0));
   end

   if set_pz1==1 %Â¿Se ha introducido valor en interfaz para calcular ZMP del sensor0?
      ZMPx1=100*((pz1*Fx1)/1000 + My1)/Fz1;
      ZMPy1=100*((pz1*Fy1)/1000 - Mx1)/Fz1;
      set(handles.ZMPx1,'String',num2str(ZMPx1));
      set(handles.ZMPy1,'String',num2str(ZMPy1));
    end
   %-------------------------------------------------------------------------

   if start01==1 %Si start01=1 es que se ha pulsado boton start01 en inertfaz para iniciar
captura de datos
       captura_datos01();

   end

   if start23==1 %Si start23=1 es que se ha pulsado boton start23 en inertfaz para iniciar
captura de datos
        captura_datos23();
    end

  toc;
  pause(0.05-toc)

end
```

```
% --- Executes on button press in OFF.
function OFF_Callback(hObject, eventdata, handles)
global on  set_fullscales0 set_fullscales1 set_fullscales2 set_fullscales3 set_pz0 set_pz1

 set_fullscales0=0;
 set_fullscales1=0;
 set_fullscales2=0;
 set_fullscales3=0;
 set_pz0=0;
 set_pz1=0;
 on=0;
 %Paramos la ejecución de jr3.cpp
 ! killall jr3

 %Apagamos PCI mediante programa reset.c
 ! ./reset

 %Paramos la ejecución de reset.c
 ! killall reset

  %Eliminamos kernel de PCI
 !rmmod jr3pci

 %Borramos memoria compartida
 borrado();


%---------------------------------Start/Finish---------------------------------------
%---------------------------------SENSOR0 y SENSOR1----------------------

% --- Executes on button press in start01.
function start01_Callback(hObject, eventdata, handles)
global start01 finish01 fx0 fy0 fz0 mx0 my0 mz0 fx1 fy1 fz1 mx1 my1 mz1
global zmpx0 zmpy0 zmpx1 zmpy1 i t01
global a b c d e f tini01 tiempo

start01=1;

tini01=tiempo;

fx0=0;
fy0=0;
fz0=0;
mx0=0;
my0=0;
mz0=0;

fx1=0;
fy1=0;
fz1=0;
```

```
mx1=0;
my1=0;
mz1=0;

a=0;
b=0;
c=0;
d=0;
e=0;
f=0;

t01=0;
i=1;

zmpx0=0;
zmpy0=0;
zmpx1=0;
zmpy1=0;


finish01=0;


%----------------------SENSOR2 y SENSOR3-----------------------------------
% --- Executes on button press in finish01.
function finish01_Callback(hObject, eventdata, handles)
 representacion_datos01();

% --- Executes on button press in start23.
function start23_Callback(hObject, eventdata, handles)
global start23 finish23
global fx2 fy2 fz2 mx2 my2 mz2 fx3 fy3 fz3 mx3 my3 mz3
global j t23
global g h k l m n tini23 tiempo

start23=1;
tini23=tiempo;

fx2=0;
fy2=0;
fz2=0;
mx2=0;
my2=0;
mz2=0;

fx3=0;
fy3=0;
fz3=0;
mx3=0;
my3=0;
```

```
mz3=0;

g=0;
h=0;
k=0;
l=0;
m=0;
n=0;

t23=0;
j=1;

finish23=0;


% --- Executes on button press in finish23.
function finish23_Callback(hObject, eventdata, handles)

  representacion_datos23();

%------------------------------set_pz0,set_pz1----------------------------
% --- Executes on button press in set_pz0.
function set_pz0_Callback(hObject, eventdata, handles)
global pz0 set_pz0
set_pz0=1;
   pz0=get(handles.pz0,'String');
   pz0=str2double(pz0);

% --- Executes on button press in set_pz1.
function set_pz1_Callback(hObject, eventdata, handles)
global pz1 set_pz1
set_pz1=1;
   pz1=get(handles.pz1,'String');
   pz1=str2double(pz1);

%-------------------------Tiempo inicial muestreo=0----------------------

% --- Executes on button press in tini01.
function tini01_Callback(hObject, eventdata, handles)
global set_tini01
set_tini01=1;

% --- Executes on button press in tini23.
function tini23_Callback(hObject, eventdata, handles)
global set_tini23
set_tini23=1;
```

```
%------------------------Sensor0, Sensor1, Sensor2, Sensor3---------------
% --- Executes on button press in sensor0.
function sensor0_Callback(hObject, eventdata, handles)
global sensor0
sensor0=1;

% --- Executes on button press in sensor1.
function sensor1_Callback(hObject, eventdata, handles)
global sensor1
sensor1=1;

% --- Executes on button press in sensor2.
function sensor2_Callback(hObject, eventdata, handles)
global sensor2
sensor2=1;

% --- Executes on button press in sensor3.
function sensor3_Callback(hObject, eventdata, handles)
global sensor3
sensor3=1;

%------------------------SET FULLSCALES-----------------------------------

% --- Executes on button press in Set_fullscales0.
function Set_fullscales0_Callback(hObject, eventdata, handles)
global set_fullscales0 on
global fsx0 fsy0 fsz0 Msx0 Msy0 Msz0  Fx0 Fy0 Fz0 Mx0 My0 Mz0 tiempo offset0

   fsx0=get(handles.fsx0,'String');
   fsx0=str2double(fsx0);

   fsy0=get(handles.fsy0,'String');
   fsy0=str2double(fsy0);

   fsz0=get(handles.fsz0,'String');
   fsz0=str2double(fsz0);

   Msx0=get(handles.Msx0,'String');
   Msx0=str2double(Msx0);

   Msy0=get(handles.Msy0,'String');
   Msy0=str2double(Msy0);

   Msz0=get(handles.Msz0,'String');
   Msz0=str2double(Msz0);
   set_fullscales0=1;

[tiempo,Fx0,Fy0,Fz0,Mx0,My0,Mz0]=leer_datos0(offset0,on,set_fullscales0,fsx0,fsy0,f
sz0,Msx0,Msy0,Msz0);
```

```
% --- Executes on button press in set_fullscales1.
function set_fullscales1_Callback(hObject, eventdata, handles)
global set_fullscales1 on
global fsx1 fsy1 fsz1 Msx1 Msy1 Msz1  Fx1 Fy1 Fz1 Mx1 My1 Mz1 tiempo offset1
  fsx1=get(handles.fsx1,'String');
  fsx1=str2double(fsx1);

  fsy1=get(handles.fsy1,'String');
  fsy1=str2double(fsy1);

  fsz1=get(handles.fsz1,'String');
  fsz1=str2double(fsz1);

  Msx1=get(handles.Msx1,'String');
  Msx1=str2double(Msx1);

  Msy1=get(handles.Msy1,'String');
  Msy1=str2double(Msy1);

  Msz1=get(handles.Msz1,'String');
  Msz1=str2double(Msz1);
  set_fullscales1=1;

[tiempo,Fx1,Fy1,Fz1,Mx1,My1,Mz1]=leer_datos1(offset1,on,set_fullscales1,fsx1,fsy1,f
sz1,Msx1,Msy1,Msz1);


% --- Executes on button press in set_fullscales2.
function set_fullscales2_Callback(hObject, eventdata, handles)
global set_fullscales2 on
global fsx2 fsy2 fsz2 Msx2 Msy2 Msz2  Fx2 Fy2 Fz2 Mx2 My2 Mz2 tiempo offset2
  fsx2=get(handles.fsx2,'String');
  fsx2=str2double(fsx2);

  fsy2=get(handles.fsy2,'String');
  fsy2=str2double(fsy2);

  fsz2=get(handles.fsz2,'String');
  fsz2=str2double(fsz2);

  Msx2=get(handles.Msx2,'String');
  Msx2=str2double(Msx2);

  Msy2=get(handles.Msy2,'String');
  Msy2=str2double(Msy2);

  Msz2=get(handles.Msz2,'String');
  Msz2=str2double(Msz2);
```

```
  set_fullscales2=1;

[tiempo,Fx2,Fy2,Fz2,Mx2,My2,Mz2]=leer_datos2(offset2,on,set_fullscales2,fsx2,fsy2,f
sz2,Msx2,Msy2,Msz2);


% --- Executes on button press in set_fullscales3.
function set_fullscales3_Callback(hObject, eventdata, handles)
global set_fullscales3 on
global fsx3 fsy3 fsz3 Msx3 Msy3 Msz3  Fx3 Fy3 Fz3 Mx3 My3 Mz3 tiempo offset3
  fsx3=get(handles.fsx3,'String');
  fsx3=str2double(fsx3);

  fsy3=get(handles.fsy3,'String');
  fsy3=str2double(fsy3);

  fsz3=get(handles.fsz3,'String');
  fsz3=str2double(fsz3);

  Msx3=get(handles.Msx3,'String');
  Msx3=str2double(Msx3);

  Msy3=get(handles.Msy3,'String');
  Msy3=str2double(Msy3);

  Msz3=get(handles.Msz3,'String');
  Msz3=str2double(Msz3);
  set_fullscales3=1;

[tiempo,Fx3,Fy3,Fz3,Mx3,My3,Mz3]=leer_datos3(offset3,on,set_fullscales3,fsx3,fsy3,f
sz3,Msx3,Msy3,Msz3);


%-----------------------------------Offsets-------------------------------

% --- Executes on button press in set_offset0.
function set_offset0_Callback(hObject, eventdata, handles)
global set_offset0

if get(hObject,'Value')
   set_offset0=1;
else
   set_offset0=0;
end

% --- Executes on button press in set_offset1.
function set_offset1_Callback(hObject, eventdata, handles)
global set_offset1
```

```
if get(hObject,'Value')
   set_offset1=1;
else
   set_offset1=0;
end

% --- Executes on button press in set_offset2.
function set_offset2_Callback(hObject, eventdata, handles)
global set_offset2

if get(hObject,'Value')
   set_offset2=1;
else
   set_offset2=0;
end

% --- Executes on button press in set_offset3.
function set_offset3_Callback(hObject, eventdata, handles)
global set_offset3

if get(hObject,'Value')
   set_offset3=1;
else
   set_offset3=0;
end


%----------------------------Gráficas SENSOR0 y SENSOR1-----------------------------

%-------------------------------Gráfica Fx---------------------------------
% --- Executes on button press in grafica_fx01.
function grafica_fx01_Callback(hObject, eventdata, handles)
global grafica_fx01
grafica_fx01=1;


%-------------------------------Gráfica Fy---------------------------------
% --- Executes on button press in grafica_fy01.
function grafica_fy01_Callback(hObject, eventdata, handles)
global grafica_fy01
grafica_fy01=1;

%-------------------------------Gráfica Fz---------------------------------
% --- Executes on button press in grafica_fz01.
function grafica_fz01_Callback(hObject, eventdata, handles)
global grafica_fz01
grafica_fz01=1;


%-------------------------------Gráfica Mx---------------------------------
```

```
% --- Executes on button press in grafica_Mx01.
function grafica_Mx01_Callback(hObject, eventdata, handles)
global grafica_Mx01
grafica_Mx01=1;


%----------------------------Gráfica My---------------------------------
% --- Executes on button press in grafica_My01.
function grafica_My01_Callback(hObject, eventdata, handles)
global grafica_My01
grafica_My01=1;

%----------------------------Gráfica Mz---------------------------------
% --- Executes on button press in grafica_Mz01.
function grafica_Mz01_Callback(hObject, eventdata, handles)
global grafica_Mz01
grafica_Mz01=1;


%----------------------------Vector fuerza---------------------------------
% --- Executes on button press in vector_fuerza01.
function vector_fuerza01_Callback(hObject, eventdata, handles)
global vector_fuerza01
vector_fuerza01=1;

%-------------------------------ZMP-------------------------------------
% --- Executes on button press in grafica_ZMP.
function grafica_ZMP_Callback(hObject, eventdata, handles)
global grafica_ZMP
grafica_ZMP=1;


%--------------------Gráficas SENSOR2 y SENSOR3--------------------------

%-------------------------------Gráfica Fx--------------------------------

%--- Executes on button press in grafica_fx23.
function grafica_fx23_Callback(hObject, eventdata, handles)
global grafica_fx23
grafica_fx23=1;


%-------------------------------GrÃ¡fica Fy--------------------------------
% --- Executes on button press in grafica_fy23.
function grafica_fy23_Callback(hObject, eventdata, handles)
global grafica_fy23
grafica_fy23=1;

% --- Executes on button press in grafica_fz23.
function grafica_fz23_Callback(hObject, eventdata, handles)
```

```
global grafica_fz23
grafica_fz23=1;


%----------------------------GrÃ¡fica Mx---------------------------------
% --- Executes on button press in grafica_Mx23.
function grafica_Mx23_Callback(hObject, eventdata, handles)
global grafica_Mx23
grafica_Mx23=1;


%----------------------------GrÃ¡fica My---------------------------------
% --- Executes on button press in grafica_My23.
function grafica_My23_Callback(hObject, eventdata, handles)
global grafica_My23
grafica_My23=1;

%----------------------------GrÃ¡fica Mz---------------------------------
% --- Executes on button press in grafica_Mz23.
function grafica_Mz23_Callback(hObject, eventdata, handles)
global grafica_Mz23
grafica_Mz23=1;

%----------------------------Vector Fuerza-------------------------------
% --- Executes on button press in vector_fuerza23.
function vector_fuerza23_Callback(hObject, eventdata, handles)
global vector_fuerza23
vector_fuerza23=1;
%--------------------------------------------------------------------------
%--------------------------------------------------------------------------


%---------------Fx0,Fy0,Fz0,Mx0,My0,Mz0---------------------------------
function Fx0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fx0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Fy0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fy0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
```

```
end


function Fz0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fz0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Mx0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mx0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function My0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function My0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Mz0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mz0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



%-------------------Fx1,Fy1,Fz1,Mx1,My1,Mz1-------------------------------

function Fx1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fx1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end


function Fy1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fy1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Fz1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fz1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Mx1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mx1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function My1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function My1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Mz1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mz1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
%-----------------Fx2,Fy2,Fz2,Mx2,My2,Mz2---------------------------------

function Fx2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fx2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Fy2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fy2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Fz2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fz2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Mx2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mx2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function My2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function My2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function Mz2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mz2_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

%-----------------Fx3,Fy3,Fz3,Mx3,My3,Mz3---------------------------------

function Fx3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fx3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Fy3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fy3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Fz3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Fz3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Mx3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mx3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function My3_Callback(hObject, eventdata, handles)
```

```
% --- Executes during object creation, after setting all properties.
function My3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

 function Mz3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Mz3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
%------------------------------------------------------------------------
%------------------------------------------------------------------------

%---------------------Full Scales-----------------------------------------

%----------------------Fsx0,Fsy0,Fsz0,Msx0,Msy0,Msz0-----------------------
function fsx0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsx0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function fsy0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsy0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function fsz0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsz0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
function Msx0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msx0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msy0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msy0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msz0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msz0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


%------------------------Fsx1,Fsy1,Fsz1,Msx1,Msy1,Msz1--------------------

function fsx1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsx1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function fsy1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsy1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
```

```
    set(hObject,'BackgroundColor','white');
end


function fsz1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsz1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msx1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msx1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msy1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msy1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msz1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msz1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


%--------------------Fsx2,Fsy2,Fsz2,Msx2,Msy2,Msz2-------------------------

function fsx2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsx2_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function fsy2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsy2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function fsz2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsz2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msx2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msx2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msy2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msy2_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msz2_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msz2_CreateFcn(hObject, eventdata, handles)
```

```
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


%-------------------------Fsx3,Fsy3,Fsz3,Msx3,Msy3,Msz3---------------------

function fsx3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsx3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function fsy3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsy3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function fsz3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function fsz3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msx3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msx3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msy3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
```

```
function Msy3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function Msz3_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function Msz3_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


%------------------------VALOR ZMP0-----------------------------------
function ZMPx0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function ZMPx0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function ZMPy0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function ZMPy0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

%-------------------------------------------------------------------------
%-----------------------VALOR ZMP 1-----------------------------------------------
function ZMPx1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function ZMPx1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
%-------------------------------------------------------------------------
```

```
%--------------------------------------------------------------------
function ZMPy1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function ZMPy1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
%--------------------------------------------------------------------


function pz0_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function pz0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function pz1_Callback(hObject, eventdata, handles)
% --- Executes during object creation, after setting all properties.
function pz1_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function set_pz0_CreateFcn(hObject, eventdata, handles)

if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

“inicializacion.m”

```
function inicializacion
global grafica_fx01 grafica_fy01 grafica_fz01 grafica_Mx01 grafica_My01
global grafica_Mz01 vector_fuerza01
global grafica_fx23 grafica_fy23 grafica_fz23 grafica_Mx23 grafica_My23
global grafica_Mz23 vector_fuerza23 grafica_ZMP
global offset0 offset1 offset2 offset3 sensor0 sensor1 sensor2 sensor3 set_pz0 set_pz1
global start01 start23
 global on

 on=0;

 offset0=0;
 offset1=0;
 offset2=0;
 offset3=0;

 set_pz0=0;
 set_pz1=0;

 start01=0;
 start23=0;

 sensor0=0;
 sensor1=0;
 sensor2=0;
 sensor3=0;

 grafica_fx01=0;
 grafica_fy01=0;
 grafica_fz01=0;
 grafica_Mx01=0;
 grafica_My01=0;
 grafica_Mz01=0;
 grafica_ZMP=0;

 vector_fuerza01=0;


 grafica_fx23=0;
 grafica_fy23=0;
 grafica_fz23=0;
 grafica_Mx23=0;
 grafica_My23=0;
 grafica_Mz23=0;

 vector_fuerza23=0;
```

“captura_datos01.m”

```
function captura_datos01

global fx0 fy0 fz0 mx0 my0 mz0 fx1 fy1 fz1 mx1 my1 mz1
global grafica_fx01 grafica_fy01 grafica_fz01 grafica_Mx01 grafica_My01
global  grafica_Mz01 grafica_ZMP
global  Fx0 Fy0 Fz0 Mx0 My0 Mz0 Fx1 Fy1 Fz1 Mx1 My1 Mz1 zmpx0 zmpy0
global ZMPx0 ZMPy0 zmpx1 zmpy1 ZMPx1 ZMPy1 set_tini01 tini01
global sensor0 sensor1 i tiempo t01 a b c d e f vector_fuerza01  set_pz0 set_pz1

if set_tini01==0
  t01(i)=tiempo;
else
   t01(i)=tiempo-tini01;
end

if grafica_fx01==1
    if sensor0==1
        fx0(i)=Fx0;
    end

    if sensor1==1
        fx1(i)=Fx1;
    end
end

if grafica_fy01==1
    if sensor0==1
        fy0(i)=Fy0;
    end

    if sensor1==1
        fy1(i)=Fy1;
    end
end

if grafica_fz01==1
    if sensor0==1
        fz0(i)=Fz0;
    end

    if sensor1==1
        fz1(i)=Fz1;
    end
end

if grafica_Mx01==1
    if sensor0==1
        mx0(i)=Mx0;
    end
```

```
    if sensor1==1
        mx1(i)=Mx1;
    end
end

if grafica_My01==1
    if sensor0==1
        my0(i)=My0;
    end

    if sensor1==1
        my1(i)=My1;
    end
end

if grafica_Mz01==1
    if sensor0==1
        mz0(i)=Mz0;
    end

    if sensor1==1
        mz1(i)=Mz1;
    end
end

if vector_fuerza01==1
    if sensor0==1
      a(i)=Fx0;
      b(i)=Fy0;
      c(i)=Fz0;
    end

    if sensor1==1
      d(i)=Fx1;
      e(i)=Fy1;
      f(i)=Fz1;
    end

end

if grafica_ZMP==1
    if set_pz0==1
       zmpx0(i)=ZMPx0;
       zmpy0(i)=ZMPy0;
    end

    if set_pz1==1
       zmpx1(i)=ZMPx1;
       zmpy1(i)=ZMPy1;
    end
```

```
end

i=i+1;
```

”captura_datos23.m”

```
function captura_datos23

global fx2 fy2 fz2 mx2 my2 mz2 fx3 fy3 fz3 mx3 my3 mz3
global  Fx2 Fy2 Fz2 Mx2 My2 Mz2 Fx3 Fy3 Fz3 Mx3 My3 Mz3
global grafica_fx23 grafica_fy23 grafica_fz23 grafica_Mx23 grafica_My23
global grafica_Mz23
global sensor2 sensor3 j tiempo t23 g h k l m n vector_fuerza23 set_tini23 tini23

if set_tini23==0
  t23(j)=tiempo;
else
    t23(j)=tiempo-tini23;
end

if grafica_fx23==1
    if sensor2==1
        fx2(j)=Fx2;
    end

    if sensor3==1
        fx3(j)=Fx3;
    end
end

if grafica_fy23==1
    if sensor2==1
        fy2(j)=Fy2;
    end

    if sensor3==1
        fy3(j)=Fy3;
    end
end

if grafica_fz23==1
    if sensor2==1
        fz2(j)=Fz2;
    end

    if sensor3==1
        fz3(j)=Fz3;
    end
end

if grafica_Mx23==1
    if sensor2==1
        mx2(j)=Mx2;
    end

    if sensor3==1
```

```
        mx3(j)=Mx3;
    end
end

if grafica_My23==1
    if sensor2==1
        my2(j)=My2;
    end

    if sensor3==1
        my3(j)=My3;
    end
end

if grafica_Mz23==1
    if sensor2==1
        mz2(j)=Mz2;
    end

    if sensor3==1
        mz3(j)=Mz3;
    end
end

if vector_fuerza23==1
    if sensor2==1
      g(j)=Fx2;
      h(j)=Fy2;
      k(j)=Fz2;
    end

    if sensor3==1
      l(j)=Fx3;
      m(j)=Fy3;
      n(j)=Fz3;
    end

end

j=j+1;
```

”representacion_datos01.m”

```
function representacion_datos01
global t01 fx0 fy0 fz0 mx0 my0 mz0 fx1 fy1 fz1 mx1 my1 mz1 sensor0 sensor1
global grafica_fx01 grafica_fy01 grafica_fz01 grafica_Mx01 grafica_My01
global grafica_Mz01
global grafica_ZMP zmpx0 zmpy0 zmpx1 zmpy1 a b c d e f vector_fuerza01
global v1 v2 v3  i set_pz0 set_pz1
global start01  set_tini01


%Representaciones de fx
   if grafica_fx01==1
         if sensor0==1
           figure(1)
           grid
           title('Sensor0:Fx')
           xlabel('Tiempo (s)')
           ylabel('Sensor0: Fx (N)')
           hold on
           plot(t01,fx0,'-*r')

            drawnow % Necesario para evitar errores de Java
            jFig = get(handle(1), 'JavaFrame');
            jFig.setMaximized(true);
         end

         if sensor1==1
           figure(7)
           grid
           title('Sensor1:Fx')
           xlabel('Tiempo (s)')
           ylabel('Sensor1: Fx (N)')
          hold on
           plot(t01,fx1,'-*r')

            drawnow % Necesario para evitar errores de Java
           jFig = get(handle(7), 'JavaFrame');
           jFig.setMaximized(true);
         end
            grafica_fx01=0;
   end

%Rpresentaciones de fy
      if grafica_fy01==1
         if sensor0==1
           figure(2)
           grid
           title('Sensor0: Fy')
           xlabel('Tiempo (s)')
           ylabel('Sensor0:  Fy (N)')
```

```
            hold on
            plot(t01,fy0,'*-r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(2), 'JavaFrame');
             jFig.setMaximized(true);
           end

           if sensor1==1
            figure(8)
            grid
            title('Sensor1: Fy')
            xlabel('Tiempo (s)')
            ylabel('Sensor1:  Fy (N)')
            hold on
            plot(t01,fy1,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(8), 'JavaFrame');
             jFig.setMaximized(true);
           end
            grafica_fy01=0;
       end

%Representaciones de fz
      if grafica_fz01==1
          if sensor0==1
             figure(3)
             grid
             title('Sensor0: Fz')
             xlabel('Tiempo (s)')
             ylabel('Sensor0:  Fz (N)')
             hold on

             plot(t01,fz0,'-*r')
              drawnow % Necesario para evitar errores de Java
              jFig = get(handle(3), 'JavaFrame');
             jFig.setMaximized(true);
           end

           if sensor1==1
              figure(9)
              grid
              title('Sensor1: Fz')
              xlabel('Tiempo (s)')
              ylabel('Sensor1:  Fz (N)')
              hold on

              plot(t01,fz1,'-*r')
```

```
            drawnow % Necesario para evitar errores de Java
            jFig = get(handle(9), 'JavaFrame');
            jFig.setMaximized(true);
        end
        grafica_fz01=0;
    end

%Representaciones de Mx
    if grafica_Mx01==1
        if sensor0==1
            figure(4)
            grid
            title('Sensor0:Mx')
            xlabel('Tiempo (s)')
            ylabel('Sensor0: Mx (N*m)')
            hold on

            plot(t01,mx0,'-*r')

            drawnow % Necesario para evitar errores de Java
            jFig = get(handle(4), 'JavaFrame');
            jFig.setMaximized(true);
        end

        if sensor1==1
            figure(10)
            grid
            title('Sensor1:Mx')
            xlabel('Tiempo (s)')
            ylabel('Sensor1: Mx (N*m)')
            hold on

            plot(t01,mx1,'*r')

            drawnow % Necesario para evitar errores de Java
            jFig = get(handle(10), 'JavaFrame');
            jFig.setMaximized(true);
        end

        grafica_Mx01=0;
    end

%Representaciones de My
    if grafica_My01==1
        if sensor0==1
            figure(5)
            grid
            title('Sensor0:My')
            xlabel('Tiempo (s)')
            ylabel('Sensor0: My (N*m)')
```

```
            hold on

            plot(t01,my0,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(5), 'JavaFrame');
             jFig.setMaximized(true);
        end

        if sensor1==1
            figure(11)
            grid
            title('Sensor1:My')
            xlabel('Tiempo (s)')
            ylabel('Sensor1: My (N*m)')
            hold on

            plot(t01,my1,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(11), 'JavaFrame');
             jFig.setMaximized(true);
        end
          grafica_My01=0;
    end

%%Representaciones de Mz
      if grafica_Mz01==1
        if sensor0==1
            figure(6)
            grid
            title('Sensor0:Mz')
            xlabel('Tiempo (s)')
            ylabel('Sensor0: Mz (N*m)')
            hold on

            plot(t01,mz0,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(6), 'JavaFrame');
             jFig.setMaximized(true);
        end

        if sensor1==1
            figure(12)
            grid
            title('Sensor1:Mz')
            xlabel('Tiempo (s)')
            ylabel('Sensor1: Mz (N*m)')
            hold on
```

```
            plot(t01,mz1,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(12), 'JavaFrame');
             jFig.setMaximized(true);
          end
           grafica_Mz01=0;
       end

%Representaciones de vector fuerza
      if vector_fuerza01==1
         i=i-1;
         if sensor0==1
            for i=i:-1:1
              v1=[0,a(i)];
              v2=[0,b(i)];
              v3=[0,c(i)];
            figure(25)
            title('Sensor0:Vector fuerza')
            xlabel('Fx (N)')
            ylabel('Fy (N)')
            zlabel('Fz (N)')
            grid
            hold on
             plot3(v1,v2,v3);

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(25), 'JavaFrame');
             jFig.setMaximized(true);
            end
          end

         if sensor1==1;
            for i=i:-1:1
            v1=[0,d(i)];
            v2=[0,e(i)];
            v3=[0,f(i)];
            figure(26)
            title('Sensor1:Vector fuerza')
            xlabel('Fx (N)')
            ylabel('Fy (N)')
            zlabel('Fz (N)')
            grid
            hold on

            plot3(v1,v2,v3);

            drawnow % Necesario para evitar errores de Java
            jFig = get(handle(26), 'JavaFrame');
```

```
            jFig.setMaximized(true);
            end
          end
           vector_fuerza01=0;
      end


% Representación trayectoria ZMP
   if grafica_ZMP==1
          if set_pz0==1
           figure(29)
           grid
           title('ZMP en apoyo simple para sensor 0')
           xlabel('ZMPx (cm)')
           ylabel('ZMPy (cm)')
           hold on

           plot(zmpx0,zmpy0,'*r')

           drawnow % Necesario para evitar errores de Java
           jFig = get(handle(29), 'JavaFrame');
           jFig.setMaximized(true);
          end
         if set_pz1==1
           figure(30)
           grid
           title('ZMP en apoyo simple para sensor1')
           xlabel('ZMPx (cm)')
           ylabel('ZMPy (cm)')
           hold on

           plot(zmpx1,zmpy1,'*b')

           drawnow % Necesario para evitar errores de Java
           jFig = get(handle(30), 'JavaFrame');
           jFig.setMaximized(true);
         end

          grafica_ZMP=0;
   end

     start01=0;
  set_tini01=0;

  sensor0=0;
  sensor1=0;
```

”representacion_datos23.m”

```
function representacion_datos23
global t23 fx2 fy2 fz2 mx2 my2 mz2 fx3 fy3 fz3 mx3 my3 mz3 sensor2 sensor3
global grafica_fx23 grafica_fy23 grafica_fz23 grafica_Mx23 grafica_My23
global grafica_Mz23
global g h k l m n  vector_fuerza23
global v1 v2 v3 j
global start23  set_tini23

%Representaciones de Fx
    if grafica_fx23==1
        if sensor2==1
          figure(13)
          grid
          title('Sensor2:Fx')
          xlabel('Tiempo (s)')
          ylabel('Sensor2: Fx (N)')
          hold on
          plot(t23,fx2,'-*r')

           drawnow % Necesario para evitar errores de Java
           jFig = get(handle(13), 'JavaFrame');
           jFig.setMaximized(true);
        end

        if sensor3==1
          figure(19)
          grid
          title('Sensor3:Fx')
          xlabel('Tiempo (s)')
          ylabel('Sensor3: Fx (N)')
         hold on
          plot(t23,fx3,'-*r')

           drawnow % Necesario para evitar errores de Java
          jFig = get(handle(19), 'JavaFrame');
          jFig.setMaximized(true);

        end
            grafica_fx23=0;
  end

%Representaciones de Fy
      if grafica_fy23==1
        if sensor2==1
          figure(14)
          grid
          title('Sensor2: Fy')
          xlabel('Tiempo (s)')
          ylabel('Sensor2:  Fy (N)')
          hold on
```

```
            plot(t23,fy2,'*-r')
             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(14), 'JavaFrame');
             jFig.setMaximized(true);
           end

           if sensor3==1
             figure(20)
             grid
             title('Sensor3: Fy')
             xlabel('Tiempo (s)')
             ylabel('Sensor3:  Fy (N)')
             hold on
             plot(t23,fy3,'-*r')

              drawnow % Necesario para evitar errores de Java
              jFig = get(handle(20), 'JavaFrame');
              jFig.setMaximized(true);
           end

           grafica_fy23=0;
       end

%Representaciones de Fz
      if grafica_fz23==1
          if sensor2==1
             figure(15)
             grid
             title('Sensor2: Fz')
             xlabel('Tiempo (s)')
             ylabel('Sensor2:  Fz (N)')
             hold on

             plot(t23,fz2,'-*r')
              drawnow % Necesario para evitar errores de Java
              jFig = get(handle(15), 'JavaFrame');
             jFig.setMaximized(true);
           end

           if sensor3==1
              figure(21)
              grid
              title('Sensor3: Fz')
              xlabel('Tiempo (s)')
              ylabel('Sensor3:  Fz (N)')
              hold on

              plot(t23,fz3,'-*r')

               drawnow % Necesario para evitar errores de Java
```

```
            jFig = get(handle(21), 'JavaFrame');
            jFig.setMaximized(true);

        end
          grafica_fz23=0;
    end

%Reprentaciones de Mx
    if grafica_Mx23==1
        if sensor2==1
            figure(16)
            grid
            title('Sensor2:Mx')
            xlabel('Tiempo (s)')
            ylabel('Sensor2: Mx (N*m)')
            hold on

            plot(t23,mx2,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(16), 'JavaFrame');
             jFig.setMaximized(true);
        end

        if sensor3==1
            figure(22)
            grid
            title('Sensor3:Mx')
            xlabel('Tiempo (s)')
            ylabel('Sensor3: Mx (N*m)')
            hold on

            plot(t23,mx3,'*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(22), 'JavaFrame');
             jFig.setMaximized(true);
        end
        grafica_Mx23=0;
    end

%Representaciones de My
    if grafica_My23==1
        if sensor2==1
            figure(17)
            grid
            title('Sensor2:My')
            xlabel('Tiempo (s)')
            ylabel('Sensor2: My (N*m)')
            hold on
```

```
            plot(t23,my2,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(17), 'JavaFrame');
             jFig.setMaximized(true);
        end

        if sensor3==1
            figure(23)
            grid
            title('Sensor3:My')
            xlabel('Tiempo (s)')
            ylabel('Sensor3: My (N*m)')
            hold on

            plot(t23,my3,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(23), 'JavaFrame');
             jFig.setMaximized(true);
        end
          grafica_My23=0;
    end

%Representaciones de Mz
      if grafica_Mz23==1
        if sensor2==1
            figure(18)
            grid
            title('Sensor2:Mz')
            xlabel('Tiempo (s)')
            ylabel('Sensor2: Mz (N*m)')
            hold on

            plot(t23,mz2,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(18), 'JavaFrame');
             jFig.setMaximized(true);
        end

        if sensor3==1
            figure(24)
            grid
            title('Sensor3:Mz')
            xlabel('Tiempo (s)')
            ylabel('Sensor3: Mz (N*m)')
            hold on
```

```
            plot(t23,mz3,'-*r')

             drawnow % Necesario para evitar errores de Java
             jFig = get(handle(24), 'JavaFrame');
             jFig.setMaximized(true);
          end
           grafica_Mz23=0;
       end

%Representaciones de vector fuerza
      if vector_fuerza23==1
         j=j-1;
         if sensor2==1
            for j=j:-1:1
              v1=[0,g(j)];
              v2=[0,h(j)];
              v3=[0,k(j)];
            figure(27)
            title('Sensor2:Vector fuerza')
            xlabel('Fx (N)')
            ylabel('Fy (N)')
            zlabel('Fz (N)')
            grid
            hold on
             plot3(v1,v2,v3);

            drawnow % Necesario para evitar errores de Java
            jFig = get(handle(27), 'JavaFrame');
            jFig.setMaximized(true);
            end
         end

         if sensor3==1;
            for j=j:-1:1
              v1=[0,l(j)];
              v2=[0,m(j)];
              v3=[0,n(j)];
            figure(28)
            title('Sensor3:Vector fuerza')
            xlabel('Fx (N)')
            ylabel('Fy (N)')
            zlabel('Fz (N)')
            grid
            hold on
             plot3(v1,v2,v3);

            drawnow % Necesario para evitar errores de Java
            jFig = get(handle(28), 'JavaFrame');
            jFig.setMaximized(true);
            end
```

```
        end
        vector_fuerza23=0;
    end

set_tini23=0;
start23=0;
sensor2=0;
sensor3=0;
```

“memoria.h”

```
 /* Fichero memoria.h, contiene información sobre la zona de memoria que se pretende compartir entre diversos procesos, será preciso incluirlo en la cabecera del fichero de aquellos procesos que se conecten a la zona de memoria común */


/*Definición de la clave de acceso a la zona de memoria común */

#define CLAVE_SHM ((key_t) 1001) //Clave de la memoria compartida

/* Estructura de datos que se pretende compartir en la zona de memoria común */
typedef struct{

      float tiempo;
       int on;

       int offset0;
       int offset1;
       int offset2;
       int offset3;

       int set_fullscales0;
       int set_fullscales1;
       int set_fullscales2;
       int set_fullscales3;

       int Fx0;
       int Fy0;
       int Fz0;
       int Mx0;
       int My0;
       int Mz0;

       int fsx0;
       int fsy0;
       int fsz0;
       int Msx0;
       int Msy0;
       int Msz0;

       int Fx1;
       int Fy1;
       int Fz1;
       int Mx1;
       int My1;
       int Mz1;

       int fsx1;
       int fsy1;
       int fsz1;
       int Msx1;
```

```
    int Msy1;
    int Msz1;

    int Fx2;
    int Fy2;
    int Fz2;
    int Mx2;
    int My2;
    int Mz2;

    int fsx2;
    int fsy2;
    int fsz2;
    int Msx2;
    int Msy2;
    int Msz2;

    int Fx3;
    int Fy3;
    int Fz3;
    int Mx3;
    int My3;
    int Mz3;

    int fsx3;
    int fsy3;
    int fsz3;
    int Msx3;
    int Msy3;
    int Msz3;
}memo;
```

"leer_datos0.cpp"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <signal.h>
#include <curses.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>

#include "mex.h"
#include "memoria.h"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[])
{
        double *a, *b, *c, *d, *e, *f, *g;
        int Fx0, Fy0, Fz0, Mx0, My0, Mz0, fsx0, fsy0, fsz0, Msx0, Msy0, Msz0,
offset0,on;
        int set_fullscales0;
        float tiempo;
        int shmid;
        memo *mem;

/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0777)) == -1){
     perror("shmget");
     exit(EXIT_FAILURE);
     }
/* Obtención del puntero a la estructura de datos compartida */

mem = (memo *)shmat(shmid,0,0);

/*Obtención de la memoria compartida de lso valores para los argumentos de salida a
pasar a "interfazJR3_dispaly"*/

tiempo=mem->tiempo; /* Se coge de la memoria compartida el valor de tiempo
guardado en la memoria por "jr3.cpp"*/
Fx0=mem->Fx0;/* Se coge de la memoria compartida el valor de Fx0 guardado en la
memoria por "jr3.cpp"*/
Fy0=mem->Fy0;/* Se coge de la memoria compartida el valor de Fx0 guardado en la
memoria por "jr3.cpp"*/
```

```
Fz0=mem->Fz0;/* Se coge de la memoria compartida el valor de Fx0 guardado en la memoria por "jr3.cpp"*/

Mx0=mem->Mx0;/* Se coge de la memoria compartida el valor de Fx0 guardado en la memoria por "jr3.cpp"*/
My0=mem->My0;/* Se coge de la memoria compartida el valor de Fx0 guardado en la memoria por "jr3.cpp"*/
Mz0=mem->Mz0;/* Se coge de la memoria compartida el valor de Fx0 guardado en la memoria por "jr3.cpp"*/


/* ----------Argumentos de salida a pasar a "inertfazJR3_display"----------*/

/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor tiempo obtenido de la memoria compartida*/
plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
a = mxGetPr(plhs[0]);
*(a) = tiempo;

/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor Fx0 obtenido de la memoria compartida*/
plhs[1]= mxCreateDoubleMatrix(1,1, mxREAL);
b = mxGetPr(plhs[1]);
*(b) = Fx0;

/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor Fy0 obtenido de la memoria compartida*/
plhs[2]= mxCreateDoubleMatrix(1,1, mxREAL);
c = mxGetPr(plhs[2]);
*(c) = Fy0;

/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor Fz0 obtenido de la memoria compartida*/
plhs[3]= mxCreateDoubleMatrix(1,1, mxREAL);
d = mxGetPr(plhs[3]);
*(d) = Fz0;

/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor Mx0 obtenido de la memoria compartida*/
plhs[4]= mxCreateDoubleMatrix(1,1, mxREAL);
e = mxGetPr(plhs[4]);
*(e) = Mx0;

/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor My0 obtenido de la memoria compartida*/
plhs[5]= mxCreateDoubleMatrix(1,1, mxREAL);
f = mxGetPr(plhs[5]);
*(f) = My0;
```

```
/*Se guarda en los argumentos de salida de "leer_datos0.cpp" el valor Mz0obtenido de la memoria compartida*/
plhs[6]= mxCreateDoubleMatrix(1,1, mxREAL);
g = mxGetPr(plhs[6]);
*(g) = Mz0;


/*----------Argumentos de entrada que pasa "inertfazJR3_display para guardar en memoria compartida----------*/

offset0=(int)mxGetScalar(prhs[0]);/*se guarda en offset0 el valor de offset0 enviado desde "interfajr3_display.m"*/
mem->offset0=offset0; /*se guarda en la memoria dicho valor de offset0*/

on=(int)mxGetScalar(prhs[1]); /*se guarda en on el valor de offset0 enviado desde "interfajr3_display.m"*/
mem->on=on;/*se guarda en la memoria dicho valor de on*/

set_fullscales0=(int)mxGetScalar(prhs[2]);/*se guarda en set_fullscales0 el valor de set_fullscales0 enviado desde "interfajr3_display.m"*/
mem->set_fullscales0=set_fullscales0;/*se guarda en la memoria dicho valor deset_fullscales0*/

fsx0=(int)mxGetScalar(prhs[3]); /*se guarda en fsx0 el valor de fsx00 enviado desde "interfajr3_display.m"*/
mem->fsx0=fsx0;/*se guarda en la memoria dicho valor de fsx0*/

fsy0=(int)mxGetScalar(prhs[4]); /*se guarda en fsy0 el valor de fsy0 enviado desde "interfajr3_display.m"*/
mem->fsy0=fsy0;/*se guarda en la memoria dicho valor de fsy0*/

fsz0=(int)mxGetScalar(prhs[5]); /*se guarda en fsz0 el valor de fsz0 enviado desde "interfajr3_display.m"*/
mem->fsz0=fsz0;/*se guarda en la memoria dicho valor de fsz0*/

Msx0=(int)mxGetScalar(prhs[6]); /*se guarda en Msx0 el valor de Msx0 enviado desde "interfajr3_display.m"*/
mem->Msx0=Msx0;/*se guarda en la memoria dicho valor de Msx0*/

Msy0=(int)mxGetScalar(prhs[7]); /*se guarda en Msy0 el valor de Msy0 enviado desde "interfajr3_display.m"*/
mem->Msy0=Msy0;/*se guarda en la memoria dicho valor de Msy0*/

Msz0=(int)mxGetScalar(prhs[8]); /*se guarda en Msz0 el valor de Msz0 enviado desde "interfajr3_display.m"*/
mem->Msz0=Msz0;/*se guarda en la memoria dicho valor de Msz0*/
}
```

”leer_datos1.cpp”

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <signal.h>
#include <curses.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>

#include "mex.h"
#include "memoria.h"


void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[])
{
        double *a, *b, *c, *d, *e, *f, *g;
        int Fx1, Fy1, Fz1, Mx1, My1, Mz1, fsx1, fsy1, fsz1, Msx1, Msy1, Msz1,
offset1,on;
        int set_fullscales1;
        float tiempo;
        int shmid;
        memo *mem;

/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0777)) == -1){
      perror("shmget");
      exit(EXIT_FAILURE);
      }
/* Obtención del puntero a la estructura de datos compartida */

mem = (memo *)shmat(shmid,0,0);


tiempo=mem->tiempo;
Fx1=mem->Fx1;
Fy1=mem->Fy1;
Fz1=mem->Fz1;

Mx1=mem->Mx1;
My1=mem->My1;
Mz1=mem->Mz1;
```

```
plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
a = mxGetPr(plhs[0]);
*(a) = tiempo;

plhs[1]= mxCreateDoubleMatrix(1,1, mxREAL);
b = mxGetPr(plhs[1]);
*(b) = Fx1;

plhs[2]= mxCreateDoubleMatrix(1,1, mxREAL);
c = mxGetPr(plhs[2]);
*(c) = Fy1;

plhs[3]= mxCreateDoubleMatrix(1,1, mxREAL);
d = mxGetPr(plhs[3]);
*(d) = Fz1;

plhs[4]= mxCreateDoubleMatrix(1,1, mxREAL);
e = mxGetPr(plhs[4]);
*(e) = Mx1;

plhs[5]= mxCreateDoubleMatrix(1,1, mxREAL);
f = mxGetPr(plhs[5]);
*(f) = My1;

plhs[6]= mxCreateDoubleMatrix(1,1, mxREAL);
g = mxGetPr(plhs[6]);
*(g) = Mz1;

offset1=(int)mxGetScalar(prhs[0]);
mem->offset1=offset1;

on=(int)mxGetScalar(prhs[1]);
mem->on=on;

set_fullscales1=(int)mxGetScalar(prhs[2]);
mem->set_fullscales1=set_fullscales1;

fsx1=(int)mxGetScalar(prhs[3]);
mem->fsx1=fsx1;

fsy1=(int)mxGetScalar(prhs[4]);
mem->fsy1=fsy1;

fsz1=(int)mxGetScalar(prhs[5]);
mem->fsz1=fsz1;

Msx1=(int)mxGetScalar(prhs[6]);
mem->Msx1=Msx1;
```

```
Msy1=(int)mxGetScalar(prhs[7]);
mem->Msy1=Msy1;

Msz1=(int)mxGetScalar(prhs[8]);
mem->Msz1=Msz1;
}
```

”leer_datos2.cpp”

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <signal.h>
#include <curses.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>

#include "mex.h"
#include "memoria.h"



void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[])
{
        double *a, *b, *c, *d, *e, *f, *g;
        int Fx2, Fy2, Fz2, Mx2, My2, Mz2, fsx2, fsy2, fsz2, Msx2, Msy2, Msz2,
offset2,on;
        int set_fullscales2;
        float tiempo;
        int shmid;
        memo *mem;

/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0777)) == -1){
      perror("shmget");
      exit(EXIT_FAILURE);
      }
/* Obtención del puntero a la estructura de datos compartida */

mem = (memo *)shmat(shmid,0,0);


tiempo=mem->tiempo;
Fx2=mem->Fx2;
Fy2=mem->Fy2;
Fz2=mem->Fz2;

Mx2=mem->Mx2;
My2=mem->My2;
```

```
Mz2=mem->Mz2;


plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
a = mxGetPr(plhs[0]);
*(a) = tiempo;

plhs[1]= mxCreateDoubleMatrix(1,1, mxREAL);
b = mxGetPr(plhs[1]);
*(b) = Fx2;

plhs[2]= mxCreateDoubleMatrix(1,1, mxREAL);
c = mxGetPr(plhs[2]);
*(c) = Fy2;

plhs[3]= mxCreateDoubleMatrix(1,1, mxREAL);
d = mxGetPr(plhs[3]);
*(d) = Fz2;

plhs[4]= mxCreateDoubleMatrix(1,1, mxREAL);
e = mxGetPr(plhs[4]);
*(e) = Mx2;

plhs[5]= mxCreateDoubleMatrix(1,1, mxREAL);
f = mxGetPr(plhs[5]);
*(f) = My2;

plhs[6]= mxCreateDoubleMatrix(1,1, mxREAL);
g = mxGetPr(plhs[6]);
*(g) = Mz2;

offset2=(int)mxGetScalar(prhs[0]);
mem->offset2=offset2;

on=(int)mxGetScalar(prhs[1]);
mem->on=on;

set_fullscales2=(int)mxGetScalar(prhs[2]);
mem->set_fullscales2=set_fullscales2;

fsx2=(int)mxGetScalar(prhs[3]);
mem->fsx2=fsx2;

fsy2=(int)mxGetScalar(prhs[4]);
mem->fsy2=fsy2;

fsz2=(int)mxGetScalar(prhs[5]);
mem->fsz2=fsz2;

Msx2=(int)mxGetScalar(prhs[6]);
```

```
mem->Msx2=Msx2;

Msy2=(int)mxGetScalar(prhs[7]);
mem->Msy2=Msy2;

Msz2=(int)mxGetScalar(prhs[8]);
mem->Msz2=Msz2;
}
```

"leer_datos3.cpp"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <fcntl.h>
#include <signal.h>
#include <curses.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <sys/msg.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>

#include "mex.h"
#include "memoria.h"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[])
{
        double *a, *b, *c, *d, *e, *f, *g;
        int Fx3, Fy3, Fz3, Mx3, My3, Mz3, fsx3, fsy3, fsz3, Msx3, Msy3, Msz3,
offset3,on;
        int set_fullscales3;
        float tiempo;
        int shmid;
        memo *mem;

/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(memo), IPC_CREAT|0777)) == -1){
     perror("shmget");
     exit(EXIT_FAILURE);
     }
/* Obtención del puntero a la estructura de datos compartida */

mem = (memo *)shmat(shmid,0,0);


tiempo=mem->tiempo;
Fx3=mem->Fx3;
Fy3=mem->Fy3;
Fz3=mem->Fz3;

Mx3=mem->Mx3;
My3=mem->My3;
Mz3=mem->Mz3;
```

```
plhs[0] = mxCreateDoubleMatrix(1,1, mxREAL);
a = mxGetPr(plhs[0]);
*(a) = tiempo;

plhs[1]= mxCreateDoubleMatrix(1,1, mxREAL);
b = mxGetPr(plhs[1]);
*(b) = Fx3;

plhs[2]= mxCreateDoubleMatrix(1,1, mxREAL);
c = mxGetPr(plhs[2]);
*(c) = Fy3;

plhs[3]= mxCreateDoubleMatrix(1,1, mxREAL);
d = mxGetPr(plhs[3]);
*(d) = Fz3;

plhs[4]= mxCreateDoubleMatrix(1,1, mxREAL);
e = mxGetPr(plhs[4]);
*(e) = Mx3;

plhs[5]= mxCreateDoubleMatrix(1,1, mxREAL);
f = mxGetPr(plhs[5]);
*(f) = My3;

plhs[6]= mxCreateDoubleMatrix(1,1, mxREAL);
g = mxGetPr(plhs[6]);
*(g) = Mz3;

offset3=(int)mxGetScalar(prhs[0]);
mem->offset3=offset3;

on=(int)mxGetScalar(prhs[1]);
mem->on=on;

set_fullscales3=(int)mxGetScalar(prhs[2]);
mem->set_fullscales3=set_fullscales3;

fsx3=(int)mxGetScalar(prhs[3]);
mem->fsx3=fsx3;

fsy3=(int)mxGetScalar(prhs[4]);
mem->fsy3=fsy3;

fsz3=(int)mxGetScalar(prhs[5]);
mem->fsz3=fsz3;

Msx3=(int)mxGetScalar(prhs[6]);
mem->Msx3=Msx3;
```

```
Msy3=(int)mxGetScalar(prhs[7]);
mem->Msy3=Msy3;

Msz3=(int)mxGetScalar(prhs[8]);
mem->Msz3=Msz3;
}
```

”reset.cpp”

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "jr3pci-ioctl.h"

int main(void) {
	six_axis_array fm;
	force_array fs;
	int ret,i,fd;

	if ((fd=open("/dev/jr3",O_RDWR)) < 0) {
		perror("Can't open device. No way to read force!");
	}

	ret=ioctl(fd, IOCTL0_JR3_RESET);

	close(fd);
}
```

“borrado.cpp”

```
#include <stdio.h>
#include <sys/shm.h>

#include "mex.h"
#include "memoria.h"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray *prhs[])
{
int shmid;
info *sensor;

/* Creación de la zona de memoria compartida */
if((shmid = shmget(CLAVE_SHM, sizeof(info), IPC_CREAT|0777)) == -1)
          {
       perror("shmget");
       exit(EXIT_FAILURE);
   }

/* Obtención del puntero a la estructura de datos compartida */
sensor = (info *)shmat(shmid,0,0);

/*Borrado de la zona de memoria compartida */
shmctl(shmid,IPC_RMID,0);

}
```

Anexo 2:

Manual JR3

JR3

DSP-BASED
FORCE SENSOR RECEIVERS

SOFTWARE AND INSTALLATION
MANUAL

***JR3,* INC.**
22 HARTER AVE.
WOODLAND, CA 95776

(530) 661-3677x4
(530) 661-3701 (fax)
jr3@jr3.com (e-mail)

5907E

Warranty Provisions

JR3, Inc. warrants this product to be in good working order for a period of one year from the date of its purchase as a new product. Should this product fail to perform properly at any time within that one year period, ***JR3,*** Inc. will, at its option, repair or replace this product at no cost except as set forth in this warranty. Replacement parts or products will be furnished on an exchange basis only. Replaced parts and/or products become the property of ***JR3,*** Inc. No warranty is expressed or implied for products damaged by accident, abuse, natural or personal disaster, or unauthorized modification.

Warranty Limitations

All warranties for this product expressed or implied, including merchantability and fitness for a purpose, are limited to a one year duration from date of purchase, and no warranties, express or implied, will apply after that period.

If this product does not perform as warranted herein, owner's sole remedy shall be repair or replacement as provided above. In no event will ***JR3,*** Inc. be liable to any purchaser for damages, lost revenue, lost wages, lost savings, or any other incidental or consequential damages arising from purchase, use, or inability to use this product, even if ***JR3,*** Inc. has been advised of such damages.

Product performance is affected by system configuration, software, the application, customer data, and operator control of the system among other factors. The specific functional implementation by users of the product will vary. Therefore, the suitability of this product for a specific application must be determined by the customer and is not warranted by ***JR3,*** Inc..

This manual is as complete and factual as possible at the time of printing, however, the information in this manual may have been updated since that time, without notice.

Copyright Notice

Printed in the United States of America.

Table of Contents

JR3's DSP-based Force Sensor Receivers
Software and Installation Manual Overview

This manual covers the setup and operation of ***JR3*** DSP-based force sensor receivers. The main part of the manual covers that information which is common to the different receivers. Information unique to a particular receiver is covered in an appendix for that receiver.

This overview contains three sections. The first discusses the layout of the manual, the second, Getting Started, makes a few suggestions about getting results quickly. The third section gives an overview of the ***JR3*** DSP architecture.

SOFTWARE AND INSTALLATION MANUAL LAYOUT

JR3's DSP-based force sensor receivers contain a dual-ported RAM that the user and ***JR3*** DSP can both read and write. The ***JR3*** DSP writes current force and moment data into this RAM. The user can then read this data from the RAM. This manual primarily consists of a description of that shared area. The data structure declarations are formatted in 'C' style code. The first section, **Data Locations and Definitions**, starts on page 7 and contains the variable declarations and descriptions. The second section, **Data Structure Definitions**, starts on page 19 and contains the data type declarations. The data declarations in both sections are shown in the style of the 'C' computer language. There is a short 'C' Language Primer starting on page 27.

The next section, Command Definitions, starts on page 29 and contains descriptions of the commands which can be given to the ***JR3*** DSP. These commands, along with various data locations are used to alter the tasks performed by the DSP. Immediately following is the Examples section on page 41.

The main part of the manual ends with a summary table for the data locations and the commands, a glossary of terms, and a brief discussion of performance issues. The appendices contain specific data on the different versions of the ***JR3***'s DSP-based force sensor receivers.

GETTING STARTED

The best place for the reader to start in this manual is with the architectural overview which follows this section. It discusses the capabilities of the ***JR3*** DSP-based force sensor receivers. Second the reader should look at the appendix which is appropriate to the particular receiver he has. This will show how to interface to the receiver from a hardware and software perspective.

If the reader has little or no experience with the 'C' computer language, he should look next at the 'C' Language Primer on page 27. After brushing up on 'C', browse the **Data Locations and Definitions** section and the **Data Structure Definitions** section. Finally the Command Definitions section and the accompanying examples show how to execute commands.

When trying to communicate with the ***JR3*** DSP for the first time, reading the copyright statement at offset 0x0040, and the **count_x** variable, at offset 0x00ef, which is discussed on page 14, provides a quick way to see if the user can read from the shared address space. Writing a value into the FX offset variable, at 0x0088, and then reading that variable back can provide a quick way to see if writing to the shared address space is successful.

ARCHITECTURAL OVERVIEW

The ***JR3*** DSP-based force sensor receivers utilize the latest technology to provide 6 degree-of-freedom (6DOF) force and moment data at very high bandwidths. Employing an Analog Devices ADSP-2105, a 10 Mips digital signal processing chip, the ***JR3*** system can provide decoupled and digitally filtered data at 8 kHz per channel. This data rate is an order of magnitude faster than previously available. The receivers have been optimized to work with ***JR3***'s force moment sensors containing onboard electronics.

JR3 DSP-based receivers are available for several interfaces, with the IBM-AT bus and VMEbus being just two examples. These boards share a common architecture. The architecture consists of a dual-ported RAM, to which the host and the ***JR3*** DSP can both read and write. This RAM allows the host to read data from the ***JR3*** DSP with very little overhead. It also allows the host to reconfigure the ***JR3*** DSP, on the fly, by writing configuration commands to the RAM.

The receiver board contains circuitry to receive the serial digital data transmission from the sensor, as well as circuitry to monitor and adjust the power supply voltage to the sensor. The automatic remote power supply adjustment means that the sensor cable requirements are very forgiving. Long, small gage wires can be used with success. This means ***JR3***'s newest sensors, with onboard electronics, no longer need stiff expensive cables.

The dual-ported address space consists of 16k 2-byte words. The first 8k of this space is RAM, while the remaining 8k consists of status registers and other features. The majority of the user activity takes place in the first 256 words of the shared address space. The location of the variables in the shared address space is documented in the Data Locations and Definitions section this manual (pg. 7). Details of reading and writing to and from the shared address space vary among the different receivers and are documented separately for each in an appendix to this manual.

The ***JR3*** DSP is used to process the raw data transmitted from the sensor. The ***JR3*** DSP performs several functions. These include: offset removal, data decoupling, saturation detection, digital low-pass filtering, force and moment vector magnitude calculation, peak detection, rate calculation, coordinate translation and rotation, and threshold monitoring.

The raw data from the sensor is passed through a decoupling matrix and offsets are removed. This process removes sensor cross-coupling as well as tare loads. One by-product of this process is that it becomes difficult to determine if the analog-to-digital converter (ADC) in the force moment transducer is saturated. If the ADC is saturated, feedback loops using the force data will become unstable. To alert the user of this condition, the ***JR3*** DSP monitors the raw sensor data and indicates when the sensor data is approaching saturation, and when it is saturated.

The decoupled data is passed through cascaded low-pass filters. Each succeeding filter is calculated 1/4 as often, and has a cutoff frequency of 1/4 of the preceding filter. The cutoff frequency of a filter is 1/16 of the sample rate for that filter. For a typical sensor with a sample rate of 8 kHz, the cutoff frequency of the first filter would be 500 Hz. The following filters would cutoff at 125 Hz, 31.25 Hz, 7.81 Hz, etc. The delay through the filter is approximately equal to:

$$\text{Delay} \cong \frac{1}{\text{Cutoff Frequency}}$$

Therefore the delay through the 500 Hz filter would be:

$$\frac{1}{500\text{Hz}} \cong 2\text{ms}$$

Since the delay varies with the frequency of the data, this value, while providing a good estimate, is not exact. For a better description of filter properties, see the accompanying graphs showing filter response.

Force and moment vector magnitudes (vectors) are calculated from each data set. Therefore, there are vectors calculated from the unfiltered decoupled data, as well as from each of the sets of filtered data. The unfiltered vectors are calculated at 1/2 the bandwidth of the unfiltered data. The vectors calculated from the filtered data are calculated at 4 times the filter cutoff frequency, or 1/4 as often as the filter is calculated.

Any single data set (8 items) can be monitored for peaks and valleys. The data is monitored at full bandwidth, and if a new minimum or maximum is detected it is stored. The minimums and maximums can be read and/or reset under user control. Any single data set (8 items) can be used for rate calculations. At a user set interval, the first derivative of a data set is also calculated and reported.

The user can apply sensor coordinate transformations to set any sensor axes origin and orientation. This will, for example, allow the user to align the force sensor axes with the user's tool axes, which can greatly simplify the mathematics needed for force control or monitoring.

The ***JR3*** DSP can be configured by the user to monitor thresholds. The ***JR3*** DSP can toggle a bit, should any data cross a user specified threshold value. These load envelopes allow the user to monitor several load trip conditions with very little overhead.

The following graphs show the characteristics of the digital filters. The response of the first 3 filters are shown. The outputs of the filters vary slightly as the frequency of the filter decreases due to the cascaded nature of the filters. The data for the filters after the third are essentially the same as filter #3 and therefore they are not shown. The only difference for filters #4 - #6 is that the frequency axis would need to be divided by four for each succeeding filter from the graphs for filter #3

Fig 1: Filter Characteristics for Filter1 with Sensor Data at 8 kHz

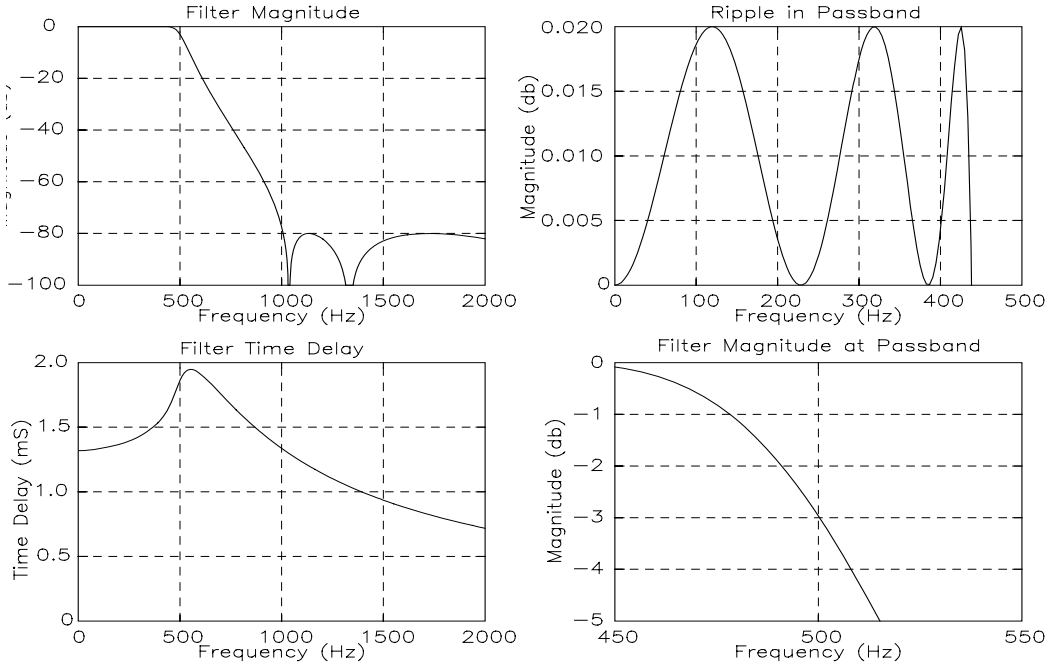


Fig 2: Filter Characteristics for Filter2 with Sensor Data at 8 kHz

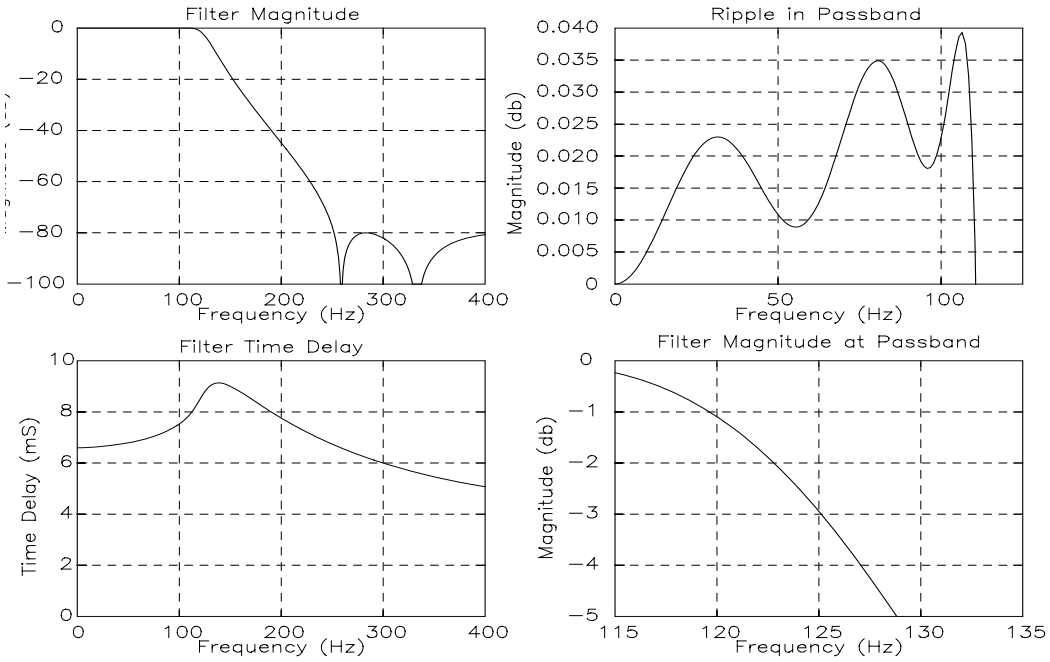
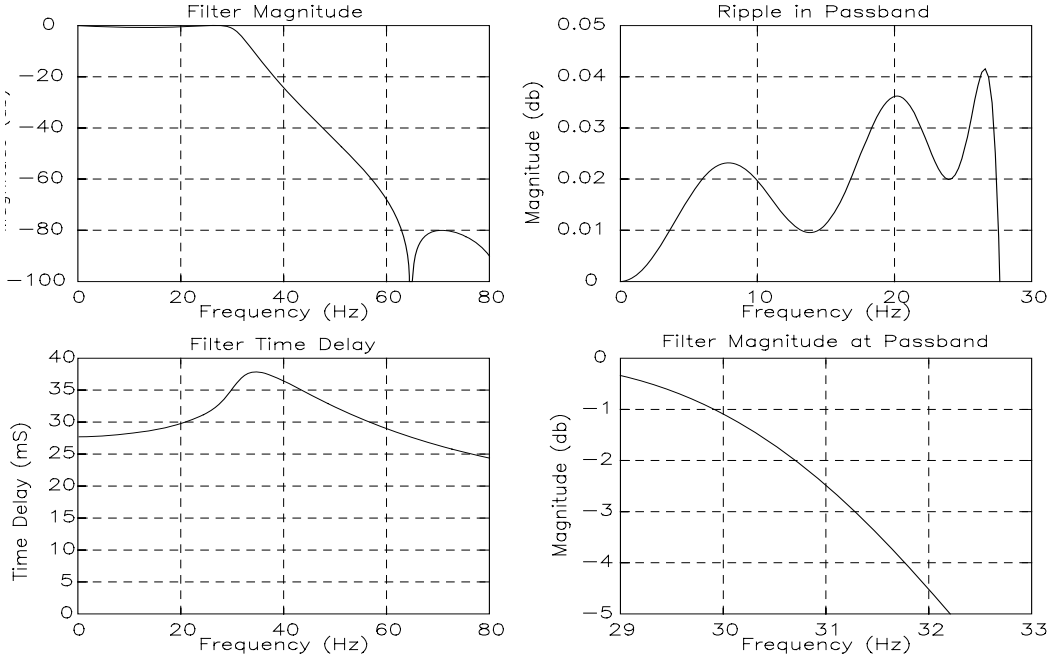


Fig 3: Filter Characteristics for Filter3 with Sensor Data at 8 kHz



JR3's DSP-based Force Sensor Receivers
Data Locations and Definitions

The following information details the memory interface of ***JR3***'s DSP-based force sensor receivers. The data structure declarations are in 'C' style code. The supporting structure declarations follow the data structure declaration. The declarations are in a courier typeface and are indented to distinguish them from their descriptions. Areas marked as reserved should not be written by the user. See the table on page 45 for a graphical layout of the variables. There is also a glossary of some of the terms on page 47.

```
/*

          For the following structure definitions:

          int:                signed 16-bit value
          unsigned int:       unsigned 16-bit value

          bit fields are shown with the lsb first.

*/
```

FORCE_SENSOR_DATA

The force_sensor_data structure shows the layout of information on the ***JR3*** DSP receiver card. The offsets shown are offsets into the ***JR3*** DSP's address space and they are for a 16-bit wide data space. Therefore, in an environment where memory is addressed 8-bits at a time, the offsets would need to be doubled.

```
struct  force_sensor_data
{
```

RAW_CHANNELS

Raw_channels is the area used to store the raw data coming from the sensor.

```
          raw_channel[16]     raw_channels        /*  offset 0x0000  */
```

COPYRIGHT

Copyright is a null terminated ASCII string containing the ***JR3*** copyright notice.

```
          int[0x0018]         copyright;          /*  offset 0x0040  */
          int[0x0008]         reserved1;          /*  offset 0x0058  */
```

SHUNTS

Shunts contains the sensor shunt readings. Some ***JR3*** sensors have the ability to have their gains adjusted. This allows the hardware full scales to be adjusted to potentially allow better resolution or dynamic range. For sensors that have this ability, the gain of each sensor channel is measured at the time of calibration using a shunt resistor. The shunt resistor is placed across one arm of the resistor bridge, and the resulting change in the output of that channel is measured. This measurement is called the shunt reading, and is recorded here. If the user has changed the gain of the sensor, and made new shunt measurements, those shunt measurements can be placed here. The ***JR3*** DSP will then scale the calibration matrix such so that the gains are again proper for the indicated shunt readings. If shunts is 0, then the sensor cannot have its gain changed. For details on changing the sensor gain, and making shunts readings, please see the sensor manual. To make these values take effect the user must call either command (5) use transform # (pg. 33) or command (10) set new full scales (pg. 38).

```
six_axis_array      shunts;             /*  offset 0x0060  */
int[2]              reserved2;          /*  offset 0x0066  */
```

DEFAULT_FS

Default_FS contains the full scale that is used if the user does not set a full scale.

```
six_axis_array      default_FS;         /*  offset 0x0068  */
int                 reserved3;          /*  offset 0x006e  */
```

LOAD_ENVELOPE_NUM

Load_envelope_num is the load envelope number that is currently in use. This value is set by the user after one of the load envelopes has been initialized.

```
int       load_envelope_num;  /*  offset 0x006f  */
```

MIN_FULL_SCALE

Min_full_scale is the recommend minimum full scale.

These values in conjunction with max_full_scale (pg. 9) helps determine the appropriate value for setting the full scales. The software allows the user to set the sensor full scale to an arbitrary value. But setting the full scales has some hazards. If the full scale is set too low, the data will saturate prematurely, and dynamic range will be lost. If the full scale is set too high, then resolution is lost as the data is shifted to the right and the least significant bits are lost. Therefore the maximum full scale is the maximum value at which no resolution is lost, and the minimum full scale is the value at which the data will not saturate prematurely. These values are calculated whenever a new coordinate transformation is calculated. It is possible for the recommended maximum to be less than the recommended minimum. This comes about primarily when using coordinate translations. If this is the case, it means that any full scale selection will be a compromise between dynamic range and resolution. It is usually recommended to compromise in favor of resolution which means that the recommend maximum full scale should be chosen.

WARNING: Be sure that the full scale is no less than 0.4% of the recommended minimum full scale. Full scales below this value will cause erroneous results.

```
six_axis_array      min_full_scale;     /*  offset 0x0070  */
int                 reserved4;          /*  offset 0x0076  */
```

TRANSFORM_NUM

Transform_num is the transform number that is currently in use. This value is set by the ***JR3*** DSP after the user has used command (5) use transform # (pg. 33).

```
int                 transform_num;      /*  offset 0x0077  */
```

MAX_FULL_SCALE

Max_full_scale is the recommended maximum full scale. See min_full_scale (pg. 9) for more details.

```
six_axis_array      max_full_scale;     /*  offset 0x0078  */
int                 reserved5;          /*  offset 0x007e  */
```

PEAK_ADDRESS

Peak_address is the address of the data which will be monitored by the peak routine. This value is set by the user. The peak routine will monitor any 8 contiguous addresses for peak values. (ex. to watch filter3 data for peaks, set this value to 0x00a8).

```
int                 peak_address;       /*  offset 0x007f  */
```

FULL_SCALE

Full_scale is the sensor full scales which are currently in use. Decoupled and filtered data is scaled so that +/- 16384 is equal to the full scales. The engineering units used are indicated by the units value discussed on page 16. The full scales for Fx, Fy, Fz, Mx, My and Mz can be written by the user prior to calling command (10) set new full scales (pg. 38). The full scales for V1 and V2 are set whenever the full scales are changed or when the axes used to calculate the vectors are changed. The full scale of V1 and V2 will always be equal to the largest full scale of the axes used for each vector respectively.

```
force_array         full_scale;         /*  offset 0x0080  */
```

OFFSETS

Offsets contains the sensor offsets. These values are subtracted from the sensor data to obtain the decoupled data. The offsets are set a few seconds (< 10) after the calibration data has been received. They are set so that the output data will be zero. These values can be written as well as read. The ***JR3*** DSP will use the values written here within 2 ms of being written. To set future decoupled data to zero, add these values to the current decoupled data values and place the sum here. The ***JR3*** DSP will change these values when a new transform is applied. So if the offsets are such that FX is 5 and all other values are zero, after rotating about Z by 90°, FY would be 5 and all others would be zero.

```
six_axis_array      offsets;            /*  offset 0x0088  */
```

OFFSET_NUM

Offset_num is the number of the offset currently in use. This value is set by the ***JR3*** DSP after the user has executed the use offset # command (pg. 34). It can vary between 0 and 15.

```
int                 offset_num;         /*  offset 0x008e  */
```

VECT_AXES

Vect_axes is a bit map showing which of the axes are being used in the vector calculations. This value is set by the ***JR3*** DSP after the user has executed the set vector axes command (pg. 37).

```
vect_bits vect_axes;          /*  offset 0x008f  */
```

FILTER0

Filter0 is the decoupled, unfiltered data from the ***JR3*** sensor. This data has had the offsets removed.

```
force_array         filter0;            /*  offset 0x0090  */
```

FILTER1 - FILTER6

These force_arrays hold the filtered data. The decoupled data is passed through cascaded low pass filters. Each succeeding filter has a cutoff frequency of 1/4 of the preceding filter. The cutoff frequency of filter1 is 1/16 of the sample rate from the sensor. For a typical sensor with a sample rate of 8 kHz, the cutoff frequency of filter1 would be 500 Hz. The following filters would cutoff at 125 Hz, 31.25 Hz, 7.813 Hz, 1.953 Hz and 0.4883 Hz.

```
force_array         filter1;            /*  offset 0x0098  */
force_array         filter2;            /*  offset 0x00a0  */
force_array         filter3;            /*  offset 0x00a8  */
force_array         filter4;            /*  offset 0x00b0  */
force_array         filter5;            /*  offset 0x00b8  */
force_array         filter6;            /*  offset 0x00c0  */
```

RATE_DATA

Rate_data is the calculated rate data. It is a first derivative calculation. It is calculated at a frequency specified by the variable rate_divisor (pg. 12). The data on which the rate is calculated is specified by the variable rate_address (pg. 12).

```
force_array         rate_data;          /*  offset 0x00c8  */
```

MINIMUM_DATA

MAXIMUM_DATA

Minimum_data & maximum_data are the minimum and maximum (peak) data values. The ***JR3*** DSP can monitor any 8 contiguous data items for minimums and maximums at full sensor bandwidth. This area is only updated at user request. This is done so that the user does not miss any peaks. To read the data, use either the read peaks command (pg. 40), or the read and reset peaks command (pg. 39). The address of the data to watch for peaks is stored in the variable peak_address (pg. 10). Peak data is lost when executing a coordinate transformation or a full scale change. Peak data is also lost when plugging in a new sensor.

```
force_array         minimum_data;       /*  offset 0x00d0  */
force_array         maximum_data;       /*  offset 0x00d8  */
```

NEAR_SAT_VALUE
SAT_VALUE

Near_sat_value & sat_value contain the value used to determine if the raw sensor is saturated. Because of decoupling and offset removal, it is difficult to tell from the processed data if the sensor is saturated. These values, in conjunction with the error and warning words (pg. 14), provide this critical information. These two values may be set by the host processor. These values are positive signed values, since the saturation logic uses the absolute values of the raw data. The near_sat_value defaults to approximately 80% of the ADC's full scale, which is 26214, while sat_value defaults to the ADC's full scale:

$$sat\_value = 32768 - 2^{(16 - \text{ADC bits})}$$

```
int       near_sat_value;               /*  offset 0x00e0  */
int       sat_value;                    /*  offset 0x00e1  */
```

RATE_ADDRESS
RATE_DIVISOR

RATE_COUNT

Rate_address, rate_divisor & rate_count contain the data used to control the calculations of the rates. Rate_address is the address of the data used for the rate calculation. The ***JR3*** DSP will calculate rates for any 8 contiguous values (ex. to calculate rates for filter3 data set rate_address to 0x00a8). Rate_divisor is how often the rate is calculated. If rate_divisor is 1, the rates are calculated at full sensor bandwidth. If rate_divisor is 200, rates are calculated every 200 samples. Rate_divisor can be any value between 1 and 65536. Set rate_divisor to 0 to calculate rates every 65536 samples. Rate_count starts at zero and counts until it equals rate_divisor, at which point the rates are calculated, and rate_count is reset to 0. When setting a new rate divisor, it is a good idea to set rate_count to one less than rate divisor. This will minimize the time necessary to start the rate calculations.

```
int                rate_address;       /*  offset 0x00e2  */
unsigned int       rate_divisor;       /*  offset 0x00e3  */
unsigned int       rate_count;         /*  offset 0x00e4  */
```

COMMAND_WORD2
COMMAND_WORD1
COMMAND_WORD0

Command_word2 through command_word0 are the locations used to send commands to the ***JR3*** DSP. Their usage varies with the command and is detailed later in the Command Definitions section (pg. 29). In general the user places values into various memory locations, and then places the command word into command_word0. The ***JR3*** DSP will process the command and place a 0 into command_word0 to indicate successful completion. Alternatively the ***JR3*** DSP will place a negative number into command_word0 to indicate an error condition. Please note the command locations are numbered backwards. (I.E. command_word2 comes before command_word1).

```
int       command_word2;               /*  offset 0x00e5  */
int       command_word1;               /*  offset 0x00e6  */
int       command_word0;               /*  offset 0x00e7  */
```

COUNT1 - COUNT6

Count1 through count6 are unsigned counters which are incremented every time the matching filters are calculated. Filter1 is calculated at the sensor data bandwidth. So this counter would increment at 8 kHz for a typical sensor. The rest of the counters are incremented at 1/4 the interval of the counter immediately preceding it, so they would count at 2 kHz, 500 Hz, 125 Hz etc. These counters can be used to wait for data. Each time the counter changes, the corresponding data set can be sampled, and this will insure that the user gets each sample, once, and only once.

```
unsigned int       count1;             /*  offset 0x00e8  */
unsigned int       count2;             /*  offset 0x00e9  */
unsigned int       count3;             /*  offset 0x00ea  */
unsigned int       count4;             /*  offset 0x00eb  */
unsigned int       count5;             /*  offset 0x00ec  */
unsigned int       count6;             /*  offset 0x00ed  */
```

ERROR_COUNT

Error_count is a running count of data reception errors. If this counter is changing rapidly, it probably indicates a bad sensor cable connection or other hardware problem. In most installations error_count should not change at all. But it is possible in an *extremely* noisy environment to experience occasional errors even without a hardware problem. If the sensor is well grounded, this is probably unavoidable in these environments. On the occasions where this counter counts a bad sample, that sample is ignored.

```
unsigned int        error_count;        /*  offset 0x00ee  */
```

COUNT_X

Count_x is a counter which is incremented every time the ***JR3*** DSP searches its job queues and finds nothing to do. It indicates the amount of idle time the ***JR3*** DSP has available. It can also be used to determine if the ***JR3*** DSP is alive. See the Performance Issues section on pg. 49 for more details.

```
unsigned int        count_x;            /*  offset 0x00ef  */
```

WARNINGS
ERRORS

Warnings & errors contain the warning and error bits respectively. The format of these two words is discussed on page 21 under the headings warnings_bits and error_bits.

```
warning_bits        warnings; /*  offset 0x00f0  */
error_bits          errors;             /*  offset 0x00f1  */
```

THRESHOLD_BITS

Threshold_bits is a word containing the bits that are set by the load envelopes. See load_envelopes (pg. 17) and thresh_struct (pg. 23) for more details.

```
int                 threshold_bits;     /*  offset 0x00f2  */
```

LAST_CRC

Last_crc is the value that shows the actual calculated CRC. CRC is short for cyclic redundancy code. It should be zero. See the description for cal_crc_bad (pg. 21) for more information.

```
int       last_CRC;           /*  offset 0x00f3  */
```

EEPROM_VER_NO
SOFTWARE_VER_NO

EEProm_ver_no contains the version number of the sensor EEProm. EEProm version numbers can vary between 0 and 255. Software_ver_no contains the software version number. Version 3.02 would be stored as 302.

```
int       eeprom_ver_no;              /*  offset 0x00f4  */
int       software_ver_no;            /*  offset 0x00f5  */
```

SOFTWARE_DAY
SOFTWARE_YEAR

Software_day & software_year are the release date of the software the ***JR3*** DSP is currently running. Day is the day of the year, with January 1 being 1, and December 31, being 365 for non leap years.

```
int       software_day;               /*  offset 0x00f6  */
int       software_year;              /*  offset 0x00f7  */
```

SERIAL_NO
MODEL_NO

Serial_no & model_no are the two values which uniquely identify a sensor. This model number does not directly correspond to the ***JR3*** model number, but it will provide a unique identifier for different sensor configurations.

```
unsigned int        serial_no;          /*  offset 0x00f8  */
unsigned int        model_no; /*  offset 0x00f9  */
```

CAL_DAY
CAL_YEAR

Cal_day & cal_year are the sensor calibration date. Day is the day of the year, with January 1 being 1, and December 31, being 366 for leap years.

```
int       cal_day;                    /*  offset 0x00fa  */
int       cal_year;          /*  offset 0x00fb  */
```

UNITS
BITS
CHANNELS

Units is an enumerated **read only** value defining the engineering units used in the sensor full scale. The meanings of particular values are discussed in the section detailing the force_units structure on page 22. The engineering units are setto customer specifications during sensor manufacture and cannot be changed by writing to Units.

Bits contains the number of bits of resolution of the ADC currently in use.

Channels is a bit field showing which channels the current sensor is capable of sending. If bit 0 is active, this sensor can send channel 0, if bit 13 is active, this sensor can send channel 13, etc. This bit can be active, even if the sensor is not currently sending this channel. Some sensors are configurable as to which channels to send, and this field only contains information on the channels available to send, not on the current configuration. To find which channels are currently being sent, monitor the Raw_time fields (pg. 19) in the raw_channels array (pg. 7). If the time is changing periodically, then that channel is being received.

```
force_units         units;              /*  offset 0x00fc  */
int                 bits;               /*  offset 0x00fd  */
int                 channels; /*  offset 0x00fe  */
```

THICKNESS

Thickness specifies the overall thickness of the sensor from flange to flange. The engineering units for this value are contained in units (pg. 16). The sensor calibration is relative to the center of the sensor. This value allows easy coordinate transformation from the center of the sensor to either flange.

```
int       thickness;                  /*  offset 0x00ff  */
```

LOAD_ENVELOPES

Load_envelopes is a table containing the load envelope descriptions. There are 16 possible load envelope slots in the table. The slots are on 16 word boundaries and are numbered 0-15. Each load envelope needs to start at the beginning of a slot but need not be fully contained in that slot. That is to say that a single load envelope can be larger than a single slot. The software has been tested and ran satisfactorily with 50 thresholds active. A single load envelope this large would take up 5 of the 16 slots. The load envelope data is laid out in an order that is most efficient for the ***JR3*** DSP. The structure is detailed later in the section showing the definition of the le_struct structure (pg. 23).

```
le_struct[0x10]     load_envelopes;     /*  offset 0x0100  */
```

TRANSFORMS

Transforms is a table containing the transform descriptions. There are 16 possible transform slots in the table. The slots are on 16 word boundaries and are numbered 0-15. Each transform needs to start at the beginning of a slot but need not be fully contained in that slot. That is to say that a single transform can be larger than a single slot. A transform is 2 * no of links + 1 words in length. So a single slot can contain a transform with 7 links. Two slots can contain a transform that is 15 links. The layout is detailed later in the section showing the definition of the transform structure (pg. 26).

```
transform[0x10]     transforms;         /*  offset 0x0200  */
```

Data Structure Definitions

RAW_CHANNEL

The raw data is stored in a format which facilitates rapid processing by the ***JR3*** DSP chip. The raw_channel structure shows the format for a single channel of data. Each channel takes four, two-byte words.

RAW_TIME

Raw_time is an unsigned integer which shows the value of the ***JR3*** DSP's internal clock at the time the sample was received. The clock runs at 1/10 the ***JR3*** DSP cycle time. ***JR3***'s slowest DSP runs at 10 Mhz. At 10 Mhz raw_time would therefore clock at 1 Mhz.

RAW_DATA

Raw_data is the raw data received directly from the sensor. The sensor data stream is capable of representing 16 different channels. Channel 0 shows the excitation voltage at the sensor. It is used to regulate the voltage over various cable lengths. Channels 1-6 contain the coupled force data Fx through Mz. Channel 7 contains the sensor's calibration data. The use of channels 8-15 varies with different sensors.

```
struct  raw_channel
{
          unsigned int        raw_time;
          int                 raw_data;
          int[2]              reserved;
};
```

FORCE_ARRAY

The force_array structure shows the layout for the decoupled and filtered force data.

```
struct  force_array
{
          int       fx;
          int       fy;
          int       fz;
          int       mx;
          int       my;
          int       mz;
          int       v1;
          int       v2;
};
```

SIX_AXIS_ARRAY

The six_axis_array structure shows the layout for the offsets and the full scales.

```
struct  six_axis_array
{
          int       fx;
          int       fy;
          int       fz;
          int       mx;
          int       my;
          int       mz;
};
```

VECT_BITS

The vect_bits structure shows the layout for indicating which axes to use in computing the vectors. Each bit signifies selection of a single axis. The V1x axis bit corresponds to a hex value of 0x0001 and the V2z bit corresponds to a hex value of 0x0020. Example: to specify the axes V1x, V1y, V2x, and V2z the pattern would be 0x002b. Vector 1 defaults to a force vector and vector 2 defaults to a moment vector. It is possible to change one or the other so that two force vectors or two moment vectors are calculated. Setting the changeV1 bit or the changeV2 bit will change that vector to be the opposite of its default. Therefore to have two force vectors, set changeV1 to 1.

```
struct  vect_bits
{
          unsigned  fx : 1;
          unsigned  fy : 1;
          unsigned  fz : 1;
          unsigned  mx : 1;
          unsigned  my : 1;
          unsigned  mz : 1;
          unsigned  changeV2 : 1;
          unsigned  changeV1 : 1;
          unsigned  reserved : 8;
};
```

WARNING_BITS

The warning_bits structure shows the bit pattern for the warning word. The bit fields are shown from bit 0 (lsb) to bit 15 (msb).

XX_NEAR_SAT

The xx_near_sat bits signify that the indicated axis has reached or exceeded the near saturation value.

```
struct  warning_bits
{
          unsigned  fx_near_sat : 1;
          unsigned  fy_near_sat : 1;
          unsigned  fz_near_sat : 1;
          unsigned  mx_near_sat : 1;
          unsigned  my_near_sat : 1;
          unsigned  mz_near_sat : 1;
          unsigned  reserved : 10;
};
```

ERROR_BITS
XX_SAT
MEMORY_ERROR
SENSOR_CHANGE

The error_bits structure shows the bit pattern for the error word. The bit fields are shown from bit 0 (lsb) to bit 15 (msb). The xx_sat bits signify that the indicated axis has reached or exceeded the saturation value. The memory_error bit indicates that a problem was detected in the on-board RAM during the power-up initialization. The sensor_change bit indicates that a sensor other than the one originally plugged in has passed its CRC check. This bit latches, and must be reset by the user.

SYSTEM_BUSY

The system_busy bit indicates that the ***JR3*** DSP is currently busy and is not calculating force data. This occurs when a new coordinate transformation, or new sensor full scale is set by the user. A very fast system using the force data for feedback might become unstable during the approximately 4 ms needed to accomplish these calculations. This bit will also become active when a new sensor is plugged in and the system needs to recalculate the calibration CRC.

CAL_CRC_BAD

The cal_crc_bad bit indicates that the calibration CRC has not calculated to zero. CRC is short for cyclic redundancy code. It is a method for determining the integrity of messages in data communication. The calibration data stored inside the sensor is transmitted to the ***JR3*** DSP along with the sensor data. The calibration data has a CRC attached to the end of it, to assist in determining the completeness and integrity of the calibration data received from the sensor. There are two reasons the CRC may not have calculated to zero. The first is that all the calibration data has not yet been received, the second is that the calibration data has been corrupted. A typical sensor transmits the entire contents of its calibration matrix over 30 times a second. Therefore, if this bit is not zero within a couple of seconds after the sensor has been plugged in, there is a problem with the sensor's calibration data.

WATCH_DOG
WATCH_DOG2

The watch_dog and watch_dog2 bits are sensor, not processor, watch dog bits. Watch_dog indicates that the sensor data line seems to be acting correctly, while watch_dog2 indicates that sensor data and clock are being received. It is possible for watch_dog2 to go off while watch_dog does not. This would indicate an improper clock signal, while data is acting correctly. If either watch dog barks, the sensor data is not being received correctly.

```
struct  error_bits
{
          unsigned  fx_sat : 1;
          unsigned  fy_sat : 1;
          unsigned  fz_sat : 1;
          unsigned  mx_sat : 1;
          unsigned  my_sat : 1;
          unsigned  mz_sat : 1;
          unsigned  reserved : 4;
          unsigned  memory_error : 1;
          unsigned  sensor_change : 1;
          unsigned  system_busy : 1;
          unsigned  cal_crc_bad : 1;
          unsigned  watch_dog2 : 1;
          unsigned  watch_dog : 1;
};
```

FORCE_UNITS

Force_units is an enumerated value defining the different possible engineering units used.

0 - lbs_in-lbs_mils:	lbs, inches * lbs, and inches * 1000
1 - N_dNm_mmX10:	Newtons, Newtons * meters * 10, and mm * 10
2 - dkgF_kgFcm_mmX10:	kilograms-force * 10, kilograms-Force * cm, and mm * 10
3 - klbs_kin-lbs_mils:	1000 lbs, 1000 inches * lbs, and inches * 1000

```
enum force_units
{
          lbs_in-lbs_mils,
          N_dNm_mmX10,
          dkgF_kgFcm_mmX10
          klbs_kin-lbs_mils
          reserved_units_4
          reserved_units_5
          reserved_units_6
          reserved_units_7
};
```

THRESH_STRUCT

Thresh_struct is the structure showing the layout for a single threshold packet inside of a load envelope. Each load envelope can contain several threshold structures.

DATA_ADDRESS

Data_address is the address of the data for that threshold. Each threshold can look at any piece of data. While the obvious filtered and unfiltered data can be monitored, it is also possible to monitor the raw data, the rate data, the counters, or the error and warning words.

THRESHOLD

Threshold is the value at which, if the data is above or below, the bits will be set. Therefore, for a greater than equal threshold (GE), if the data value is above the threshold value, the bit pattern will be OR'ed into the threshold variable at 0x00f2.

BIT_PATTERN

Bit_pattern contains the bits that will be set if the threshold value is met or exceeded.

```
struct    thresh_struct
{
          int       data_address;
          int       threshold;
          int       bit_pattern;
};
```

LE_STRUCT

Le_struct is the structure showing the layout of a load envelope packet. This structure shows 4 thresholds, but the thresholds can in fact be laid end to end for as many thresholds as needed. If there are more than four thresholds the load envelope will overlap the succeeding load envelope. This is acceptable as long as the user realizes this and does not try to use the succeeding load envelope. The thresholds need to be arranged with the greater than or equal thresholds (GE) first and the less than or equal (LE) thresholds next.

LATCH_BITS

Latch_bits is a bit pattern which shows which bits the user wants to latch. The latched bits will not be reset once the threshold which set them is no longer true. In that case, the user must reset them using the reset_bit command.

NUMBER_OF_GE_THRESHOLDS

NUMBER_OF_LE_THRESHOLDS

These values specify how many GE thresholds there are and how many LE thresholds there are. The GE thresholds are first, and are followed by the LE thresholds.

```
struct    le_struct
{
          int                 latch_bits;
          int                 number_of_ge_thresholds;
          int                 number_of_le_thresholds;
          thresh_struct[4]    thresholds;
          int                 reserved  ;
};
```

Fig 4: Load Envelope Structure

Address	data	
0	latch Pattern	Pattern of bits which latch
1	no of GE	number of >= thresholds
2	no of LE	number of <= thresholds
3	data addr	data address for GE #1
4	threshold	threshold for GE #1
5	bit pattern	bit pattern for GE #1
6	data addr	data address for GE #2
7	threshold	threshold for GE #2
8	bit pattern	bit pattern for GE #2
	...	
Ge#*3	data addr	data address for last GE
ge#*3+1	threshold	threshold for last GE
ge#*3+2	bit pattern	bit pattern for last GE
ge#*3+3	data addr	data address for LE #1
ge#*3+4	threshold	threshold for LE #1
ge#*3+5	bit pattern	bit pattern for LE #1
	...	
	data addr	data address for last LE
	threshold	threshold for last LE
	bit pattern	bit pattern for last LE

LE - **L**ess than or **E**qual threshold
GE - **G**reater than or **E**qual threshold

LINK_TYPES

Link_types is an enumerated value showing the different possible transform link types. The end transform packet is put at the end of the transform chain. The translate and rotate types are used to translate the sensor axes origin and orientation. The negate all axes type makes all axes negative. It is used to convert from the default robot point of view to the tool point of view.

0 - end transform packet
1 - translate along X axis (TX)
2 - translate along Y axis (TY)
3 - translate along Z axis (TZ)
4 - rotate about X axis (RX)
5 - rotate about Y axis (RY)
6 - rotate about Z axis (RZ)
7 - negate all axes

```
enum link_types
{
          end_x_form,
          tx,
          ty,
          tz,
          rx,
          ry,
          rz,
          neg
};
```

TRANSFORM

Transform is a structure which is used to describe a transform. A transform is made up of successive links. The links can be in any order. Each link is two words. The first word is the type, and the second word is the amount to transform. The types are detailed in the description for link_types (pg. 25). The amount to translate is specified in the engineering length units described by the **units** variable (pg. 16). The amount to rotate is scaled such that:

$$\text{amount} = \text{degrees} \cdot \frac{32768}{180} \quad \text{or} \quad \text{amount} = \text{radians} \cdot \frac{32768}{\pi}$$

These units work very nicely, because a 16 bit integer maps exactly into 0-2pi range. The translation units however, can cause a problem if extraordinarily long translations are necessary, because the absolute value of the translation must be ≤ 32767. The solution, is to break a very long translation into two or more shorter translations in the same direction but with lengths ≤ 32767. The negate all axes type must have a non-zero amount specified, the value specified for amount is not significant.

```
struct links
{
          link_types          link_type
          int                 link_amount;
};

struct transform
{
          links[8]            link;
};
```

Fig 5: Transform Structure

Address	data	
0	link # 1 - type	Link #1 packet
1	link # 1 - amount	
2	link # 2 - type	Link #2 packet
3	link # 2 - amount	
	...	
(n-1)*2	link #n - type	Link #n packet
(n-1)*2+1	link #n - amount	
N*2	0	end_x_form packet
n*2+1	don't care	

‘C’ Language Primer

The data declarations in the “**Data Locations and Definitions**“ and “**Data Structure Definitions**“ sections, are shown in the style of the ‘C’ computer language. ‘C’ was chosen because it is the closest thing to a universal language currently in wide spread usage. But for those fortunate enough to have never worked with ‘C’, this is a short tutorial in reading ‘C’ style data declarations.

The symbols /* and */ are used as the begin and end comment delimiters. The semicolon ;, is used as a statement terminator, and the comma is used to separate items in a list. Hexadecimal numbers are declared by appending a prefix of 0x. So to indicate the hexadecimal value of 16 we would write 0x0010. Like wise, 157 would be 0x009d. When declaring a variable, the variable’s type is listed first, followed by the name of the variable. So to declare a variable of type int, with the name sat_value, the following would be used:

```
int sat_value;
```

Int is the basic signed integer type in ‘C’. The size of an int can vary depending on the machine for which the ‘C’ compiler has been written. In our case the natural size for an int is 16 bits, since the ADSP-2105 processes information 16 bits at a time. Therefore an int can take values ranging from -32768 to 32767. The unsigned int data type is the same size as an int, but has no sign. Therefore, the following:

```
unsigned int countx;
```

would declare a variable called countx, which could take on the value of from 0 to 65535.

‘C’ has the ability to bundle data together into structures. A struct declaration defines a data structure. After defining a struct, a variable can be declared with the type of the struct, and then individual fields of the struct can be accessed. To declare a force array structure the following could be used:

```
struct force_structure
{
          int fx;
          int fy;
          int fz;
};
```

This declares a struct named force_structure with three int fields named: fx, fy, and fz. These fields would be arranged in memory with fx at the lowest, or first, address and fz at the highest, or last, address. To define a variable using a struct, the struct name is used for the variable type in the variable declaration.

```
force_structure filter0;
```

would declares a variable called filter0, of type force_structure. Using the earlier declaration of force_structure, it would have three fields named: fx, fy and fz.

A data array in 'C' is declared by appending [x] to the variable type in a normal data declaration. The x, in [x] indicates the number of elements contained in the array. So,

```
force_structure[0x0010] force_data;
```

would indicate we had a variable called force_data, which contained 16 (0x0010 in hexadecimal) elements of type force_structure. Therefore force_data would consist of 48 (16 * 3) ints. This can be thought of as 16 different sets of force data laid end to end in memory.

Finally 'C' has a data structure type called bit fields, which allows the easy manipulation of individual bits or groups of bits in a variable. Bit fields are indicated by appending :x to the element name in a struct definition. The x in :x indicates the number of bits used by the element. 'C' allows the bit order to specified by the compiler writer, we define them to be declared from lsb to msb. That is if you were to encode our bit field as a binary number, the first bit field listed would have the least weight, and the last bit the most.

```
struct test_bits
{
          fx_bit : 1;
          mx_bit : 1;
          reserved : 14;
}
```

This declares a struct where the lsb is to be considered the fx_bit, the second bit is the mx_bit, and the top 14 bits are reserved.

Command Definitions

EXAMPLE COMMAND (0)

Calling Parameters:

Data_1	This is the name of the data to modify prior to setting command_word_0
Command_Word_0	0x0000

Returned Variables:

Data_2	This is the name of the data which is modified by this command.
Command_Word_0	0 (Command_Word_0 is set to zero to indicate that the command has successfully completed)

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The example command shows the layout of the other commands in this section. For the calling parameters and the returned variables, the first column shows the name of the variable. The second column shows the value which the variable is set to or the value which is to be returned.

If the user needed to execute this example command, the first step would for the user to modify Data_1 as needed. The user would then write the command number to Command_Word_0. In this case that would be 0x0000. The user would then monitor Command_Word_0 until it was changed to 0 by the ***JR3*** DSP. At that point the command would have been successfully executed and the user could examine the value returned in Data_2.

MEMORY READ (1)

Calling Parameters:

Command_Word_1	Address to read
Command_Word_0	0x0100

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The memory read command is of no use in standard operation. Memory can be read directly by the host. This command is only used for testing purposes, and to read the internal memory of the DSP.

MEMORY WRITE (2)

Calling Parameters:

Command_Word_2	Data to write
Command_Word_1	Address to write
Command_Word_0	0x0200

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The memory write command is used to write to a section of the DSP memory. It is only needed if there is a chance the DSP may modify the location at the same time the user is trying to write to it. This is primarily true of the latching bits of the warning, error and threshold bit pattern locations. This command also returns the value that was present in the location before it wrote to the location.

BIT SET (3)

Calling Parameters:

Command_Word_2	Bit map of bits to set
Command_Word_1	Address in which bits will be set
Command_Word_0	0x0300

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The bit set command is used to easily set single or multiple bits of a word in memory. It is only needed if there is a chance the DSP may modify the location at the same time the user is trying to write to it. This is primarily true of the latching bits of the warning, error, and threshold bit pattern locations. This command also returns the value that was present in the location before it wrote to the location.

BIT RESET (4)

Calling Parameters:

Command_Word_2	Bit map of bits to reset
Command_Word_1	Address in which will be reset
Command_Word_0	0x0400

Returned Variables:

Command_Word_2	Data read from address
Command_Word_0	0

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The bit reset command is used to easily reset single or multiple bits of a word in memory. It is only needed if there is a chance the DSP may modify the location at the same time the user is trying to write to it. This is primarily true of the latching bits of the warning, error and threshold bit pattern locations. This command also returns the value that was present in the location before it wrote to the location.

USE TRANSFORM # (5)

Calling Parameters:

transforms	Setup appropriate slot of the transform table
Command_Word_0	0x050?,

where "?" is the transform slot to use and can be 0x0 to 0xf.

Returned Variables:

Command_Word_0	0
Transform_Num	new current transform number

Execution time:

3 - 10 mS (average)

The time this command takes is directly related to the number of links in the transform. The time is approximately 2.25 ms + 0.75 ms for each link.

Notes:

Before executing the use transform command, a transform must be setup in the appropriate slot in the transform table. If the user does not require multiple transforms to be stored in local memory, just place the transform into slot 0 (@ 0x200) and then place 0x0500 into command_word_0. This command takes several milliseconds to execute, so the system_busy bit will be set in the error_bits word while it is executing. Also, as with the other commands, command_word_0 will be reset to 0 at the completion of this command.

USE OFFSET # (6)

Calling Parameters:

Command_Word_0	0x060?,

where " ? " is the offset number to use and can be 0x0 to 0xf.

Returned Variables:

Command_Word_0	0
offset_num	new current offset # from command_word_0
offsets	set to new value from offset table

Execution time:

150 µS	(average)
225 µS	(maximum)

Notes:

The ***JR3*** DSP is capable of storing 16 different offsets. This might be needed when switching to and from different tooling, or might be needed for different tooling orientations. The use offset command sets the current offset #. It also loads the offsets from the offset table. Subsequent set offsets commands or reset offsets commands will write the new offsets into this entry of the table. If the user does not require multiple offsets to be stored locally, then this command would never be called and offset #0 would always default to the working offset. If the user wants to set the current offset # without loading the offsets from the table, just write the desired offset # directly to the current offset_num location (@ 0x008e).

SET OFFSETS (7)

Calling Parameters:

offsets	set to desired offsets
Command_Word_0	0x0700

Returned Variables:

Command_Word_0	0
current offset table entry	values from the offsets variable

Execution time:

150 µS	(average)
225 µS	(maximum)

Notes:

The set offsets command takes the values from the offsets variable (@ 0x0088) and uses it for the current sensor offsets as well as storing them in the offset table under the current offset_num entry. This command is not really necessary because simply changing the values in the offsets variable will achieve the same results. The command is desirable because the ***JR3*** DSP can take up to 2 milliseconds to notice that the values have changed, and to start using them. This command speeds up the process considerably. If a 2 ms delay in changing offsets is not a problem, then simply placing the new offsets into the offsets array will suffice.

RESET OFFSETS (8)

Calling Parameters:

Command_Word_0	0x0800

Returned Variables:

Command_Word_0	0
offsets	set so that the output data will be 0
current offset table entry	values from the offsets variable

Execution time:

150 µS	(average)
225 µS	(maximum)

Notes:

The reset offsets command takes the data from filter2 and then sets the offsets such that this data will be 0 after the offset change. The command updates the values in the offsets variable (@ 0x0088) as well as storing the offsets in the offset table under the current offset_num entry. This command is not really necessary because the user can calculate and change the offset values as needed, but this command makes the chore somewhat simpler.

SET VECTOR AXES (9)

Calling Parameters:

Command_Word_0	0x09??

where "??" is a bit map describing the axes to use.
V1x = 1, V1y = 2, V1z = 4, V2x = 8, V2y = 16, V2z = 32
V1 is force vector, V2 is moment vector = 0
V1 is force vector, V2 is force vector = 64
V1 is moment vector, V2 is moment vector = 128

Returned Variables:

Command_Word_0	0
vect_axes	same bit map passed in command_word_0
full_scale	full scales for V1 and V2 are set equal to the largest component axis.

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The set vector axes command allows the user to set which axes are used for calculating the V1 (vector 1) and V2 (vector 2) vector resultants. There are two vectors calculated. They can be calculated from either forces or moments. Normally V1 is calculated from the forces and V2 from the moments. But it is also possible to set them both to be either force vectors or moment vectors.

SET NEW FULL SCALES (10)

Calling Parameters:

full_scale of fx-mz	Set to desired full scale
Command_Word_0	0x0A00

Returned Variables:

Command_Word_0	0

Execution time:

3 - 10 mS	(average)

The time the set new full scales command takes is directly related to the number of links in the current transform. The time is approximately 3.4 ms + 0.75 ms for each link.

Notes:

The set new full scales command takes several milliseconds to execute, so the system_busy bit will be set in the error_bits word while it is executing. Also, as with the other commands, command_word_0 will be reset to 0 at the completion of this command.

This command calculates a new coordinate transform as part of its normal processing. So if the user has changed the variable transform_num (pg. 9) or has changed the current slot of the transform table, this command will alter the coordinate transform. This side effect can be useful, or an unexpected surprise, so please be aware of it.

Please refer to the section on minimum_full_scale (pg. **9**) for details on choosing an appropriate full scale.

READ AND RESET PEAKS (11)

Calling Parameters:

peak_address	points to data to watch for peaks
Command_Word_0	0x0B00

Returned Variables:

Command_Word_0	0
minimums	minimum values seen since last peak reset
maximums	maximum values seen since last peak reset

Execution time:

40 µS	(average)
125 µS	(maximum)

Notes:

The read and reset peaks command takes the peak values which the ***JR3*** DSP stores internally, and copies those values to the variables minimums (@ 0x00d0) and maximums (@ 0x00d8). It then resets the internal minima and maxima to be the same as the current data. The minima and maxima operate in this fashion so that the user will not miss any minima or maxima.

\

READ PEAKS (12)

Calling Parameters:

peak_address	points to data to watch for peaks
Command_Word_0	0x0C00

Returned Variables:

Command_Word_0	0
minimums	minimum values seen since last peak reset
maximums	maximum values seen since last peak reset

Execution time:

40 μS	(average)
125 μS	(maximum)

Notes:

The read peaks command takes the peak values which the ***JR3*** DSP stores internally, and copies those values to the variables minimums (@ 0x00d0) and maximums (@ 0x00d8). The minima and maxima operate in this fashion so that the user will not miss any minima or maxima. To reset the minima and maxima see command (11) read and reset peaks (pg. 39).

Examples

Following are some examples of using ***JR3***'s DSP-based force sensor receivers. The examples are shown in a pseudo-code. We will use ## to indicate that the remainder of a line is a comment. We need to define several functions:

```
readData (addr)
```

which returns the 16 bit value which is at addr in the DSP's address,

```
writeData (addr, data)
```

which writes data to addr in the DSP's address space and,

```
write     stuff to write
```

which displays '*stuff to write*' to the user's terminal.

EXAMPLE #1 - GET DSP SOFTWARE VERSION

This example will write the currently executing DSP software version number. This value is stored at offset 0x00f5.

```
Write 'The Software Version # is '
Write (readData (0x00f5) / 100)
```

EXAMPLE #2 - GET SCALED FORCE DATA FOR FX

This example retrieves the full scale for force X (@ 0x0080) and applies it to the current force X data from filter #2 (@ 0x00a0).

```
Write 'Current FX from filter 2 is '
Write (readData (0x00a0) / 16384 * readData (0x0080))
```

EXAMPLE #3 - RESET OFFSETS

This example uses the reset offsets command (pg. 36) to make the sensor data of all axes equal to zero. This has the effect of removing the tare weight. It writes the value 0x0800, which is command 8, to the 0x0e7 location, which is command_word0 (pg. 13)

```
writeData (0x00e7, 0x0800)
```

EXAMPLE #4 - SET OFFSET, FORCE Z

This example sets the offset for the FZ axis. It does this by reading the current FZ data from filter 2 (@ 0x00a2) and the current FZ offset (@ 0x008a), and summing them. This value summed with the desired offset is then written back into the FZ offset location.

```
writeData (0x008a, readData (0x00a2) + readData (0x008a) - 20)
```

When setting offsets using the method shown above we need to rely on the DSP noticing that we have changed the offset. The DSP only looks for this change in offsets one in every 16 samples from the sensor. So if the sensor is sending data at 8 kHz, the DSP may take up to 2 mS to notice the changed offset. For many applications 2 mS is too slow, so we need to invoke command (7) set offsets (pg. 35) after writing the desired offsets to the DSP's address space to speed up the process. We do this by writing 0x0700, which is command 7, to the 0x0e7 location, which is command_word0

```
## first set the FZ offset to 20
writeData (0x008a, readData (0x00a2) + readData (0x008a) - 20)

## now write the rest of the offsets if desired
          ...

## finally, invoke the set offsets command
writeData (0x00e7, 0x0700)
```

EXAMPLE #5 - SET VECTOR AXES

This example will set the axes used for the vector calculations. By default the vectors V1 and V2 are a force and a moment vector respectively, each using all three axes. To change the axes used for V1 and V2 use command (9) set vector axes (pg. 37). The following example will set up V1 to use Fx and Fy and V2 to use Fx, Fy and Fz. It does this by writing 0x097b to the 0x0e7 location, which is command_word0 (pg. 13)

```
writeData (0x00e7, 0x097b)
```

The value 0x097b is constructed by adding the command value 0x0900 to the axes bits needed for V1 (1 = V1x, 2 = V1y), the axes bits needed for V2 (8 = V2x, 16 = V2y, 32 = V2z) and the bit needed to make V2 uses forces instead of moments (64 = V1 and V2 forces). 1+2+8+16+32+64 = 123 (0x007b).

EXAMPLE #6 - USE COORDINATE TRANSFORMATION

This example shows how to use a simple coordinate transform. This transform will simply rotate about the Y axis by -180°. We will be using transform table entry 0. First write the transform type for rotate Y (5) to the first entry in the transform table (0x0200). Then write the amount to transform, -180°, to the second entry. This value is encoded as -32768 as discussed in the transform section on page 26. To indicate that this is the last transform in the list, the list is terminated with a 0. Finally use command (5) use transform # (pg. 33).

```
writeData (0x0200, 5)         ## rotate about Y axis
writeData (0x0201, -32768)    ## rotate -180 degrees
writeData (0x0202, 0)         ## last transform in list
writeData (0x00e7, 0x0500)    ## use transform #0
```

EXAMPLE #7 - USE COMPLEX COORDINATE TRANSFORMATION

This example shows how to use a complex coordinate transform. This transform will start with a rotation about the Z axis by 45º. We will be using transform table entry 2. First write the transform type for rotate Z (6) to the first entry in the transform table (0x0220). Then write the amount to transform, 45º, to the second entry. This value is encoded as 8192 as discussed in the transform section on page 26. Second, we will translate along the Z axis by ½ of the sensor thickness. This will put the coordinate axis on the flange of the sensor. Then, to indicate that this is the last transform in the list, the list is terminated with a 0. Finally use command (5) use transform # (pg. 33) to effect the new coordinate transform.

```
writeData (0x0220, 6)          ## rotate about Z axis
writeData (0x0221, 8192)       ## rotate 45 degrees
writeData (0x0222, 3)          ## translate along Z axis

## translate ½ sensor thickness
writeData (0x0223, readData (0x00ff) / 2)

writeData (0x0224, 0)          ## last transform in list
writeData (0x00e7, 0x0502)     ## use transform #2
```

EXAMPLE #8 - USE A LOAD ENVELOPE

This example shows how to prepare and use a load envelope with multiple thresholds. We will be using 5 thresholds on two different axes. We will setup a positive and negative limit on Fx and Fy of 1/4 of full scale. We will setup these thresholds to each trigger two bits. We will latch one of the bits. We will also setup a threshold on the force vector at 1/2 full scale. Finally we read the threshold bits and then reset them.

```
## Start by putting the appropriate value into load envelope #2
writeData (0x0120, 0xff00)    ## latch top 8 bits
writeData (0x0121, 3)         ## we have 3 GE thresholds
writeData (0x0122, 2)         ## we have 2 LE thresholds

## first Greater or Equal threshold
writeData (0x0123, 0x0090)    ## addr of filter0 fx
writeData (0x0124, 4096)      ## 1/4 of full-scale
writeData (0x0125, 0x0101)    ## use bits 0 and 8

## second Greater or Equal threshold
writeData (0x0126, 0x0091)    ## addr of filter0 fy
writeData (0x0127, 4096)      ## 1/4 of full-scale
writeData (0x0128, 0x0202)    ## use bits 1 and 9

## last Greater or Equal threshold
writeData (0x0129, 0x0096)    ## addr of filter0 v1
writeData (0x012a, 8192)      ## 1/2 of full-scale
writeData (0x012b, 0x1010)    ## use bits 4 and 12

## first Less or Equal threshold
writeData (0x012c, 0x0090)    ## addr of filter0 fx
writeData (0x012d, -4096)     ## -1/4 of full-scale
writeData (0x012e, 0x0404)    ## use bits 2 and 10

## starting with this threshold we will spill into slot #3
## last Less or Equal threshold
writeData (0x012f, 0x0091)    ## addr of filter0 fy
writeData (0x0130, -4096)     ## -1/4 of full-scale
writeData (0x0131, 0x0808)    ## use bits 3 and 11

writeData (0x006f, 2)         ## we are using LE slot #2

write 'The current threshold status is '
write (readData (0x00f2))

## Now reset the latch bits
writeData (0x00e5, 0xff00)    ## reset top 8 bits
writeData (0x00e6, 0x00f2)    ## address of threshold bits
writeData (0x00e7, 0x0400)    ## command 4 - reset bits
```

Table 1: Summary of *JR3* DSP Data locations

	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00	ch0time	ch0data			ch1time	ch1data		
0x08	ch2time	ch2data			ch3time	ch3data		
0x38	chEtime	chEdata			chFtime	chFdata		
0x40	'C '	'o '	'p '	'y '	'r '	'i '	'g '	'h '
0x48	't '	' '	'J '	'R '	'3 '	', '	' '	'I '
0x50	'n '	'c '	' '	'1 '	'9 '	'9 '	'4 '	0
0x58								
0x60	shunt fx	shunt fy	shunt fz	shunt mx	shunt my	shunt mz		
0x68	def fs fx	def fs fy	def fs fz	def fs mx	def fs my	def fs mz		load env #
0x70	min fs fx	min fs fy	min fs fz	min fs mx	min fs my	min fs mz		xForm #
0x78	max fs fx	max fs fy	max fs fz	max fs mx	max fs my	max fs mz		peak addr
0x80	fs fx	fs fy	fs fz	fs mx	fs my	fs mz	fs v1	fs v2
0x88	ofs fx	ofs fy	ofs fz	ofs mx	ofs my	ofs mz	ofs #	vect axes
0x90	f0 fx	f0 fy	f0 fz	f0 mx	f0 my	f0 mz	f0 v1	f0 v2
0x98	f1 fx	f1 fy	f1 fz	f1 mx	f1 my	f1 mz	f1 v1	f1 v2
0xa0	f2 fx	f2 fy	f2 fz	f2 mx	f2 my	f2 mz	f2 v1	f2 v2
0xa8	f3 fx	f3 fy	f3 fz	f3 mx	f3 my	f3 mz	f3 v1	f3 v2
0xb0	f4 fx	f4 fy	f4 fz	f4 mx	f4 my	f4 mz	f4 v1	f4 v2
0xb8	f5 fx	f5 fy	f5 fz	f5 mx	f5 my	f5 mz	f5 v1	f5 v2
0xc0	f6 fx	f6 fy	f6 fz	f6 mx	f6 my	f6 mz	f6 v1	f6 v2
0xc8	rate fx	rate fy	rate fz	rate mx	rate my	rate mz	rate v1	rate v2
0xd0	min fx	min fy	min fz	min mx	min my	min mz	min v1	min v2
0xd8	max fx	max fy	max fz	max mx	max my	max mz	max v1	max v2
0xe0	near sat	sat	rate addr	rate div	rate count	comm 2	comm 1	comm 0
0xe8	count 1	count 2	count 3	count 4	count 5	count 6	errors	count x
0xf0	warning	error	threshold	crc	rom ver #	ver no	ver day	ver year
0xf8	serial	model	cal day	cal year	units	bits	chans	thickness

0x100-0x1ff - Load envelope table (threshold monitoring), 16 entries
0x200-0x2ff - Transform table (translations and rotations), 16 entries

Description of table entrys, see text for full description and missing entries:

ch0time, ch0data	time last data for channel 0 was received, last data received for raw channel 0
shunt fx,...	shunt reading for fx channel
def fs fx,...	sensor default full scale
min fs fx,...	min full scale, at which the data will not have the lsb zero filled
max fs fx,...	max full scale, at which the data will not have the lsb truncated
fs fx,...	full scale value for fx, when fx = 16384 this is the equivalent engineering units
load env #	number of currently active load envelope
xForm #	number of the transform currently in use
peak addr	addr of the data used in finding the maxima and minima
ofs fx,...	current offset value for fx
ofs #	number of the offset currently in use
vect axes	bit map for the axes which are being used for calculating the vectors
f0 fx,f0 fy,...	decoupled, unfiltered data
f1 fx,...	fx from filter 1
rate fx,...	rate calculation for fx
min fx,..., max fx,...	minimum peak (valley) value for fx, maximum peak value for fx
near sat, sat	raw value which sets near sat bit in warning word, and sat bit in the error word
rate addr	address of data used for calculating the rate data
rate div	rate divisor, the number of samples between rate calculations
rate count	this counter counts up to rate div, and then the rates are calculated
comm2,...	command word 2, 1 and 0. Area used to send commands to ***JR3*** DSP
count1,...	counter for filter #1, 1 count = 1 filter iteration
errors	a count of data reception errors
warning, error, threshold	warning word, error word, threshold monitoring word (load envelopes)
rom ver no	version no. of data stored in sensor EEProm
ver no, ver day	software version # that the ***JR3*** DSP is running, ***JR3*** DSP software release date
serial, model	sensor serial number, and sensor model number
cal day	last calibration date of the sensor
units	engineering units of full scale, 0 is lbs, in-lbs and in*1000, 1 is Newtons, ...
bits	number of bits in sensor ADC
chans	bit map of channels the sensor is capable of sending
thickness	the thickness of the sensor

Table 2: Summary of *JR3* DSP commands

Command Name	Command Words				Other Data Sent	Data Returned
	0 hi	0 lo	1	2		
mem read	1	-	addr	-	-	cw2 = data read
mem write	2	-	addr	data	-	cw2 = data read
bit set	3	-	addr	bits	-	cw2 = data read
bit reset	4	-	addr	bits	-	cw2 = data read
use transform	5	xform #	-	-	xform table	transform_num
use offset	6	ofs #	-	-	-	ofs_num, offsets
set offset	7	-	-	-	offsets	-
reset offset	8	-	-	-	-	offsets
set vec axes	9	axes	-	-	-	vect_axes
set full-scale	10	-	-	-	full_scale fx-mz	-
read and reset peak	11	-	-	-	-	min and max
read peak	12	-	-	-	-	min and max

Column Descriptions:

Command Name is the name of the command.

Command Words are located at 0x00e5 - 0x00e7.

0 hi is the upper byte of command word 0.

0 lo is the lower byte of command word 0.

1 and 2 are command word 1 and 2.

Other Data Sent indicates which other data needs to be setup before command word 0 is written

Data Returned indicates which data is affected by the command

For all commands, write command_word_0 last. Command_word_0 will return 0 or negative to indicate command completion. For further details of each command see text.

Glossary of Terms

0x0406	Hexadecimal notation for the decimal number 1030
ADC	Analog to Digital Converter: This is the device in the sensor which converts the analog voltage produced by the strain gages into a digital signal used by the ***JR3*** DSP.
Bit Fields	Data fields in which individual bits have meanings distinct from other bits in the same data field.
Byte	An eight bit data value.
Fx, Fy, Fz	Force X, Force Y and Force Z
GE	Greater than or Equal to.
LE	Less than or Equal to.
lsb	Least Significant Bit: bit 0, or the bit with a binary value of 1. The lower case distinguishes it from LSB. (see LSB).
LSB	Least Significant Byte: In a multi-byte data value, this signifies the byte with the least value. To use a decimal example, the least significant digit of 1234 is 4. The uppercase distinguishes it from lsb.
LSW	Least Significant Word: see LSB
msb	Most Significant Bit: see lsb
MSB	Most Significant Byte: see LSB
MSW	Most Significant Word: see LSB
Mx, My, Mz	Moment X, Moment Y and Moment Z
Robot point of view	The ***JR3*** force moment sensor axes are oriented so that they form a right hand rule when the tool side of the sensor is considered to be fixed and loads are applied to the robot side. We call this the robot point of view.
Tool point of view	The tool point of view is when the robot side of the force sensor is considered to be fixed, and loads are applied to the tool side of the sensor. When using this point of view the ***JR3*** sensor will appear to have a left hand coordinate system. Use the negate transform type to adjust this if needed.
V1, V2	Abbreviations for Vector 1 and Vector 2.
WORD	A 16-bit data value.

Performance Issues

The ***JR3*** DSP-based force sensor receiver uses the very high performance ADSP-2105 digital signal processor. But even this chip has a finite amount of processing power. Therefore some compromises have been made in computing some of the data values. This section details the frequency at which the different data items are calculated. Detailing these timing factors should help the user interface to the data.

All calculation frequencies are slaved to the data rate of the sensor. The standard sensor data rate is 8 kHz. For the rest of this discussion calculation frequencies will be discussed in terms of fractions of sensor bandwidth. So 1/2 bandwidth would be 4 kHz for an 8 kHz sensor.

Sensor data is decoupled and the offsets are removed at full sensor bandwidth. Filter1 and Peak data are also calculated at full sensor bandwidth. The Rate data is calculated at the user specified fraction of sensor bandwidth of 1/x, where x can be 1 to 65536. So with a fraction of 1/1 the rate data could be calculated at full sensor bandwidth. The saturation status bits are monitored at full sensor bandwidth, while all other data is calculated at less than sensor bandwidth.

The load envelope thresholds are monitored at 1/4 sensor bandwidth. The filtered data is calculated such that each filter is calculated at 1/4 the bandwidth of the preceding filter.

Filter 1 - 1/1 bandwidth
Filter 2 - 1/4 bandwidth
Filter 3 - 1/16 bandwidth
Filter 4 - 1/64 bandwidth
Filter 5 - 1/256 bandwidth
Filter 6 - 1/1024 bandwidth

The vectors for each data set are calculated from that data set. The vectors themselves are not filtered, but they are calculated from filtered data.

Vectors for Filter 0 - 1/2 bandwidth
Vectors for Filter 1 - 1/4 bandwidth
Vectors for Filter 2 - 1/16 bandwidth
Vectors for Filter 3 - 1/64 bandwidth
Vectors for Filter 4 - 1/256 bandwidth
Vectors for Filter 5 - 1/256 bandwidth
Vectors for Filter 6 - 1/1024 bandwidth

Because the user has the ability to impose a varying processing load on the DSP, it is possible for the user to hang the DSP if too much is asked from it. The two primary culprits are the rates command, and the load envelopes. A secondary influence is the processing time taken away from the DSP by the host when the host reads data. If the host is sitting in a tight timing loop waiting for command_word_0 to change to 0, or is otherwise making heavy usage of the DSP's local bus by reading or writing to it, the DSP can be slowed down. This will not be an issue for doing normal data reads, but a very fast host in a tight loop could cause problems.

The variable count_x (pg. 14) can be used to gage how busy the processor is. Count_x increments every time the DSP searches its job queues and finds nothing to do. To be sure that all tasks are being completed, we should see at least one count on count_x for each data packet. Therefore, for an 8 kHz sensor data rate, we would want to see count_x changing at least 8000 times per second.

As of software version 2.0, each count of count_x represents approximately 5.4 uS of free time. So for every count above the 8,000 Hz baseline, we can impose approximately 5.4 uS more load per second on the processor. Each iteration of the rate routine takes approximately 3.3 uS, while each additional load envelope threshold takes approximately 0.6 uS.

Example 1: For an 8 kHz sensor, the configuration under test is showing count_x changing at about 20,000 Hz. Therefore we have approximately (20,000 Hz - 8,000 Hz) * 5.4 uS = 64.8 mS/second available. How many more load envelope thresholds can we process? Each threshold takes 0.6 uS * 2,000 Hz = 1.2 mS / second, so 64.8 / 1.2 = 54 more thresholds could, at best, be calculated.

Example 2: For an 8 kHz sensor, a test at ***JR3*** of a system with the rates calculating at 1/800 bandwidth and with no load envelope thresholds, showed count_x changing at 30,500 Hz. So if the rates were changed to 1/1 bandwidth how many load envelope thresholds could be calculated? (30,500 Hz - 8,000 Hz) * 5.4 uS = 121.5 mS / second available. We need 799/800 * 8,000 more rates at 3.3 uS per rate gives 26.4 mS / second for the rates, which leaves 121.5 - 26.4 = 95.1 mS / second. From Ex. 1 the thresholds are 1.2 mS / second, which gives 95.1/1.2 = 79 thresholds maximum.

These examples are not meant to recommend the use of more than the 50 thresholds suggested on pg. 17. But, with the proper testing in the user's application, more thresholds could probably be used, especially if rate calculations are not needed.

JR3's DSP-based Force Sensor Receiver Card for the VMEbus

This appendix describes the setup and operation of the Force Sensor Receiver for the VMEbus. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a single wide (4H) double high (6U), VMEbus backplane slot. The receiver uses only the P1 connector. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the front panel. Also included is an indicator lamp which shows green if the sensor is plugged in and powered up. The lamp shows red if the sensor is not plugged in, and is off if the sensor power is off.

POWER

The receiver requires no external power. It draws power directly from the VMEbus. The board uses the following voltages and currents:

5V - 870 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly 100 mA from the -12V.

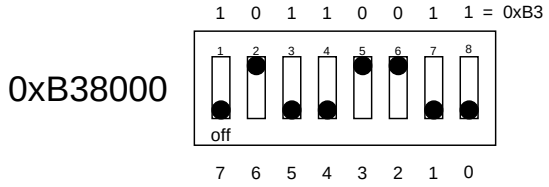
SOFTWARE INTERFACE

The ***JR3*** DSP receiver's address space is mapped directly into the VME A24 address space. Therefore, the offsets given in the section describing the memory interface need to be multiplied by 2 and then added to the base address of the card as selected in the next section. The ***JR3*** DSP receiver is then simply accessed as memory on the VMEbus.

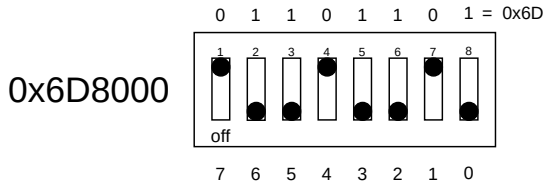
ADDRESS SELECTION

The VMEbus receiver goes into a 32k-byte block in the VME's A24 address space. The board can reside in the upper half of any 64k-byte block. The receiver adheres to the VMEbus A24 and D16 specifications with one exception: The board will respond to 8-bit data accesses with a Bus Error. The board responds to 4 address modifier codes: 0x39, 0x3a, 0x3d, 0x3e.

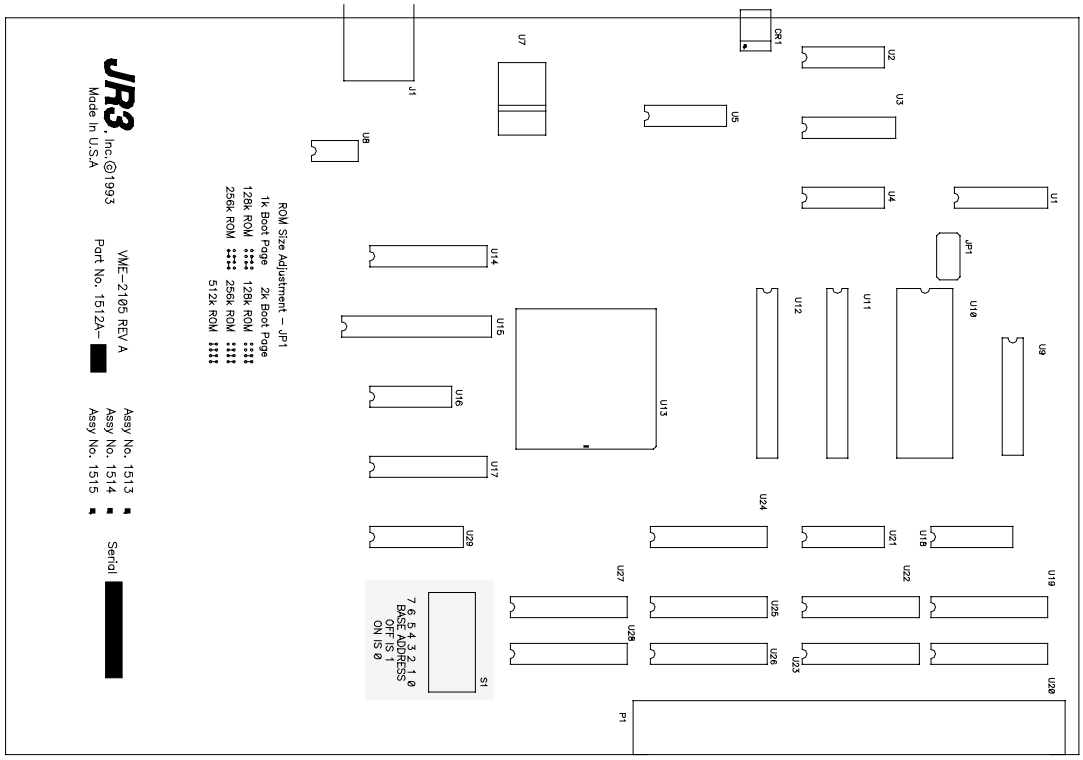
The architecture of the board from the VMEbus perspective, is 32k bytes laid out as 16k 2-byte words. The base address is selected by dip switches on the receiver. The dip switches are labeled on the silk screen as 7 through 0. These switches correspond to address bits A23 through A16. These switches make the bit take the value of 1 when they are off and 0 when they are on. Example: to set a base address of 0xB38000 the switches would be set to:



to the set the address to 0x6D8000:



Outline Drawing Showing Address DIP Switches for *JR3* VMEbus, DSP-based receiver card



JR3's DSP-based Force Sensor Receiver Card for the ISA (IBM-AT) bus.

This appendix describes the setup and operation of the Force Sensor Receiver for the ISA bus. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a 16 bit connector on the ISA bus. The receiver uses both the 62-pin and the 36-pin connectors. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the back panel.

POWER

The receiver requires no external power. It draws power directly from the ISA bus. The board uses the following voltages and currents:

5V - 650 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly as much as 100 mA from the -12V.

SOFTWARE INTERFACE

The architecture of the DSP receiver board from the ISA bus perspective is two 16-bit wide registers. The address register is at I/O addresses zero and one relative to the base address. The data register is at addresses two and three relative to the base address. Therefore if the board base address is 0x314, then the 16-bit port at 0x314 is the address port, while the 16-bit port at 0x316 is the data port.

To read a data value from the DSP's address space, first write the address of the desired data to the address port, then read the data from the data port. Writing data is done in an analogous manner.

The ISA receiver board also has a block transfer ability. To read or write data from consecutive addresses, simply write the address of the first data to the address register, then read or write each succeeding data value from or to the data register. The only caveat with this operation is that only the bottom 8 bits of the address will update. Therefore if the address 0xfe were written to the address register, the first data read would be from address 0xfe, the second from 0xff, and third from 0x00.

The following routine will read data from the board. The address should be in register AX, the data will be returned in AX.

```
MOV       dx, 0x314 ; Board base address is 0x314
OUT       dx, ax              ; Write addr to addr reg
ADD       dx, 2               ; Data reg is 2 above addr reg
IN        ax, dx              ; Read data from board
```

The following is a 'C' version of data read.

```
#include  <dos.h>
#define   baseAddr  0x314
int       getData(int addr)
{
          outport(baseAddr, addr);
          return inport(baseAddr+2);
}
```

The following routine will write data to the board. The address should be in register AX, the data in register BX.

```
MOV       dx, 0x314 ; Board base address is 0x314
OUT       dx, ax              ; Write addr to addr reg
ADD       dx, 2               ; Data reg is 2 above addr reg
MOV       ax, bx              ; Put the data into ax
OUT       dx, ax              ; Write data to the board
```

The following is a 'C' version of data write.

```
#include  <dos.h>
#define   baseAddr 0x314
int       putData(int addr, int data)
{
          outport(baseAddr, addr);
          outport(baseAddr+2, data);
}
```

The following will read a block of data from the board. The address of the beginning of the data block should be in register AX. Registers ES:DI should point to a buffer where the data block will be stored. Register CX should be the number of words of data to transfer. This code uses the INSW instruction which is not available on an 8086/8088. But since this board must go into a 16-bit expansion slot, the processor should be an 80286 or above.

```
MOV       dx,0x314            ; Board base address is 0x314
AND       ax,$3FFF            ; AX should only have 14 bits
OUT       dx,ax               ; Write the address
ADD       dx,2                ; Point to data port
PUSHF                         ; Save flags
CLD                           ; Clear the direction flag
CLI                           ; Hold interrupts
REP       INSW                ; Do the data transfer
POPF                          ; Restore the flags
```

The following will write a block of data to the board. The destination address of the data block should be in register AX. Registers DS:SI should point to a buffer where the data block is stored. Register CX should be the number of words of data to transfer. This code uses the OUTSW instruction which is not available on an 8086/8088. But since this board must go into a 16-bit expansion slot, the processor should be an 80286 or above.

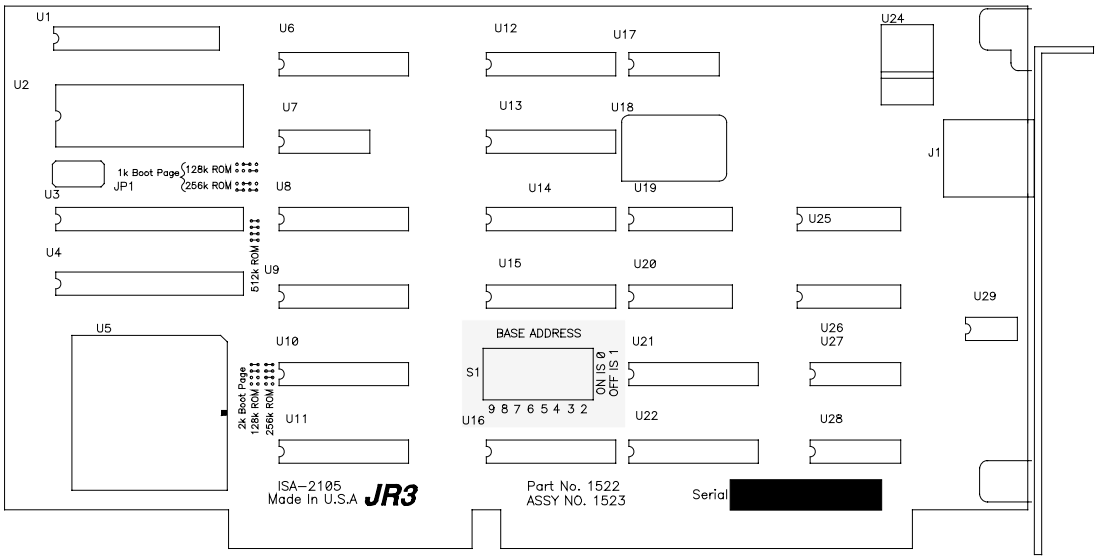
```
MOV       dx,0x314          ; Board base address is 0x314
AND       ax,$3FFF          ; AX should only have 14 bits
OUT       dx,ax             ; Write the address
ADD       dx,2              ; Point to data port
PUSHF                       ; Save flags
CLD                         ; Clear the direction flag
CLI                         ; Hold interrupts
REP       OUTSW             ; Do the data transfer
POPF                        ; Restore flags
```

ADDRESS SELECTION

The ISA bus receiver goes into 4 consecutive I/O addresses. The board can reside in any 4 byte wide space from addresses 0x000 to 0x3ff. The base address is selected by dip switches on the receiver. The dip switches are labeled on the silk screen as 9 through 2. These switches correspond to address bits A9 through A2. These switches make the bit take the value of 1 when they are off and 0 when they are on. Example: to set a base address of 0x314 the switches would be set to:

to the set the address to 0x260:

Outline Drawing Showing Address DIP Switches for *JR3* ISA bus, DSP-based Receiver Card



JR3's DSP-based Force Sensor Receiver Card for the Stäubli UNIVAL Robot Controller.

This appendix describes the setup and operation of the Force Sensor Receiver for the Stäubli UNIVAL controller. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a standard Stäubli UNIVAL controller slot. This slot is similar to a VMEbus slot except that the card is 220mm tall instead of 160 mm tall. The receiver uses only the P1 connector. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the front panel. Also included is an indicator lamp which shows green if the sensor is plugged in and powered up. The lamp shows red if the sensor is not plugged in, and is off if the sensor power is off.

POWER

The receiver requires no external power. It draws power directly from the Stäubli UNIVAL controller. The board uses the following voltages and currents:

5V - 870 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly 100 mA from the -12V.

SOFTWARE INTERFACE

The ***JR3*** DSP receiver's address space is mapped directly into the Stäubli UNIVAL A24 address space. Therefore, the offsets given in the section describing the memory interface need to be multiplied by 2 and then added to the base address of the card as selected in the section on address selection (pg. 59). The ***JR3*** DSP receiver is then simply accessed as memory on the Stäubli UNIVAL bus.

The following are some example programs to show basic interfacing to the ***JR3*** DSP receiver for the Stäubli UNIVAL bus. They are excerpted from the file getdata.val available from ***JR3***.

This program must be run to initialize the software environment for the ***JR3*** DSP receiver.

```
; To set a new base address change the two most significant
; digits in the following hex number to agree with the value
; set by the base address switch on the JR3 DSP receiver board.
; i.e. if switches set to ^Hb0
    !JR3.base = ^Hb08000
; Enable 'Z' instruction set
    parameter terminal = ^H50001
```

This program shows how to read a single data value from the ***JR3*** DSP receiver:

```
.program JR3.read.port (!data.addr, data)
    data = setr (!zpeekw (!JR3.base+!data.addr*2))
    IF data > 32767 THEN
        data = data - 65536
    END
```

This program shows how to write a single data value to the ***JR3*** DSP receiver:

```
.program JR3.write.port (!data.addr, data)
    local !data
    IF data > 32767 THEN
        !data = 32767
    ELSE
        IF data < -32768 THEN
            !data = 32768
        ELSE
            IF data < 0 THEN
                !data = !seti (data + 65536)
            ELSE
                !data = !seti (data)
            END
        END
    END
    zpokew (!JR3.base+!data.addr*2) = !data
```

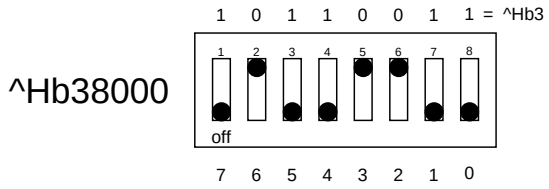
The following code will return the current value of FX in engineering units

```
    CALL JR3.read.port (^H90, tmp1)
    CALL JR3.read.port (^H80, tmp2)
    fx = tmp1 / 16384 * tmp2
```

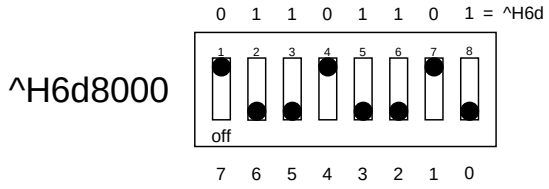
ADDRESS SELECTION

The Stäubli UNIVAL receiver goes into a 32k-byte block in the Stäubli UNIVAL's A24 address space. The board can reside in the upper half of any 64k-byte block. The receiver adheres to the VMEbus A24 and D16 specifications with one exception: The board will respond to 8-bit data accesses with a Bus Error. The board responds to 4 address modifier codes: 0x39, 0x3a, 0x3d, 0x3e.

The architecture of the board from the Stäubli UNIVAL controller perspective is 32k bytes laid out as 16k 2-byte words. The base address is selected by dip switches on the receiver. The dip switches are labeled on the silk screen as 7 through 0. These switches correspond to address bits A23 through A16. These switches make the bit take the value of 1 when they are off and 0 when they are on. Example: to set a base address of ^Hb38000 the switches would be set to:

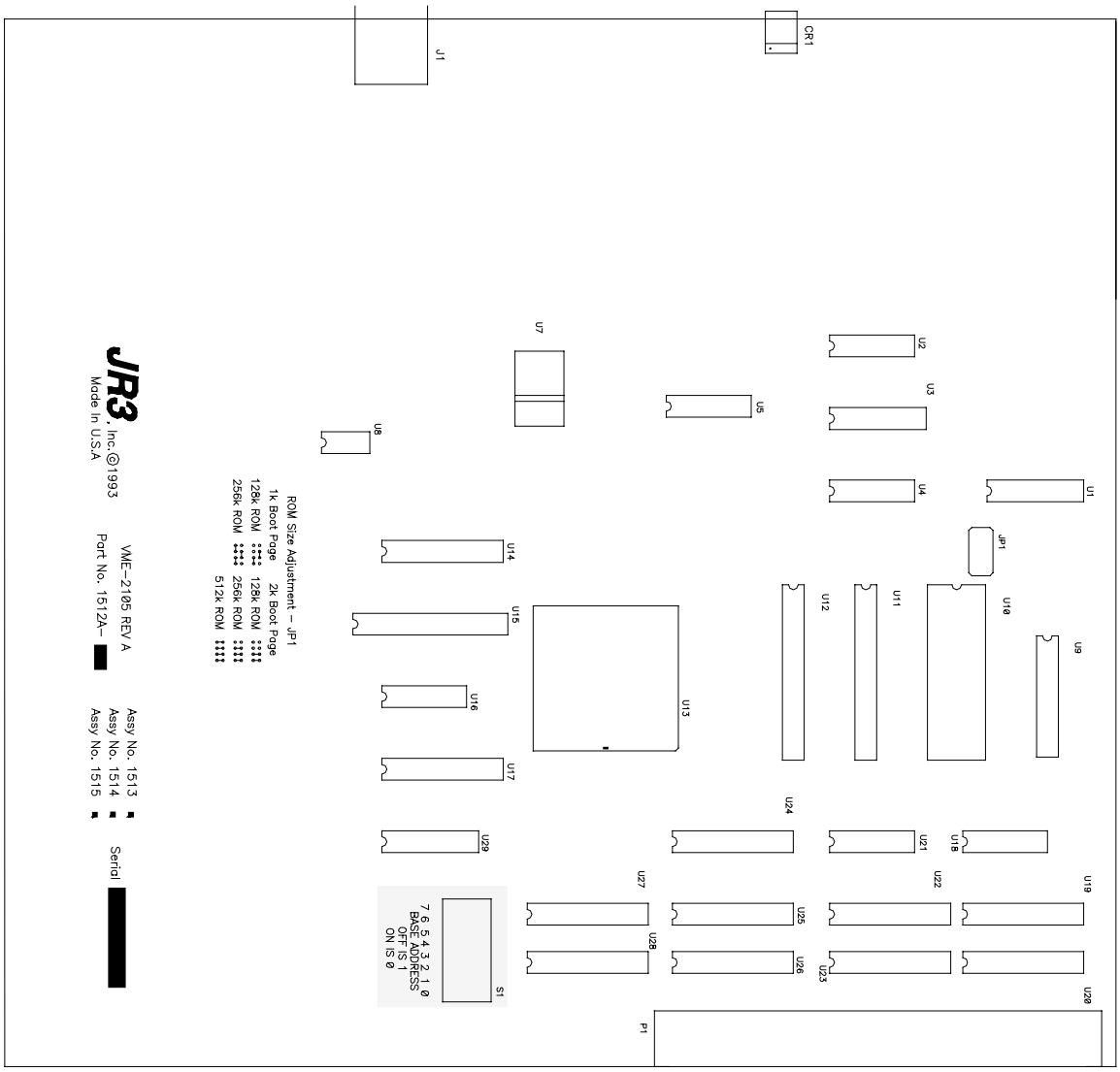


to the set the address to ^H6d8000:



Choosing a base address is somewhat more complicated than setting it. It is impossible for ***JR3*** to recommend a suitable address for all applications but if the user has no knowledge of his memory map, ***JR3*** suggests trying ^Hb38000.

Outline Drawing Showing Address DIP Switches for *JR3* Stäubli UNIVAL, DSP-based receiver card



JR3's DSP-based Force Sensor Receiver Card for the PCI bus

This appendix describes the setup and operation of the Force Sensor Receiver for the PCI bus. It covers only those aspects of the card that differ from other members of ***JR3***'s DSP-based force sensor receiver family.

FORM FACTOR

The receiver card plugs into a standard length PCI Bus slot. It is a 5V, 33 MHz, 32-Bit PCI card. It does **not** support 3.3 Volt or 66 MHz operation. The receiver interfaces to the sensor through an 8-pin (RJ-45) modular jack on the back panel. Also included are two indicator led. The first led shows green if the DSP has successfully booted. The second led shows green if the sensor is plugged in and powered up.

POWER

The receiver requires no external power. It draws power directly from the PCI bus. The board uses the following voltages and currents:

5V - 870 mA typical
12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

The sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly 100 mA from the -12V.

SOFTWARE INTERFACE

The ***JR3*** PCI DSP receiver's address space is mapped directly into the PCI data address space. The 16 bit wide DSP memory is mapped into the 32 bit wide PCI bus. Therefore, the offsets given in the section describing the memory interface need to be multiplied by 4 and then added to the base address of the card + 0x6000 hex. The ***JR3*** DSP receiver is then simply accessed as memory on the PCI bus.

ADDRESS SELECTION

The PCI bus is a plug and play bus, there is no user configuration necessary. The receiver goes into a 512k-byte block in the PCI's address space. The receiver responds to 4 command types: 0x6 Memory Read, 0x7 Memory Write, 0xa Configuration Read and 0xb Configuration Write.

CODE DOWNLOAD

The ***JR3*** PCI DSP receiver's card code must be downloaded from the host. After any reset, and before the card can be used, it must have code downloaded. C Source to perform this function is available from ***JR3***

Anexo 3:

Manual para la instalación sensores fuerza/par de JR3 inc.

JR3

INSTALLATION MANUAL
FOR
FORCE - TORQUE SENSORS
WITH INTERNAL ELECTRONICS

***JR3,* Inc.**
22 Harter Ave.
Woodland, CA 95776

5977-

TABLE OF CONTENTS

CHAPTER 1
SENSOR OVERVIEW

The ***JR3*** sensor is a monolithic aluminum (optionally stainless steel or titanium) device containing analog and digital electronics systems. Foil strain gages sense the loads imposed on the sensor. The strain gage signals are amplified and combined to become analog representations of the force loads on the three axes and the moments or torques about the three axes. In most models, the analog data is converted to digital form by electronic systems contained within the sensor. Some models optionally provide analog output instead of digital.

Sensors are produced in a wide variety of load ratings and bolt patterns. The physical size of the sensor varies, depending on factors such as force and moment ratings and required mounting dimensions. A drawing of your specific sensor and a detailed specification sheet is provided with your sensor.

The axes on standard ***JR3*** sensors are oriented with the X and Y axes in the plane of the sensor body, and the Z axis perpendicular to the X and Y axes. The reference point for all loading data is the geometric center of the sensor. When viewed from the Robot Side of the sensor the forces and moments are related by the Right Hand Rule.

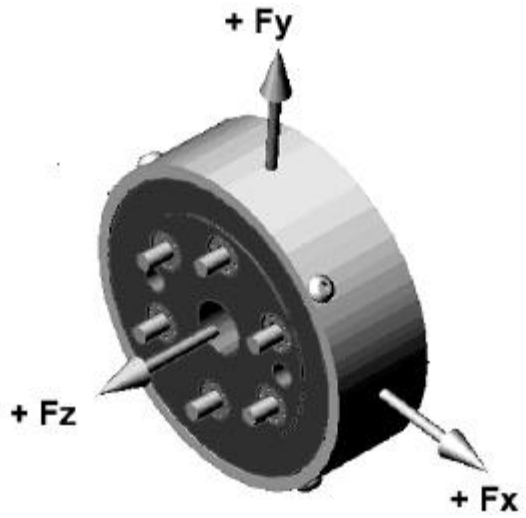


Fig. 1–1
SENSOR AXIS ORIENTATION FROM ROBOT SIDE

CHAPTER 2
INSTALLATION

This chapter provides general instructions for installing the force - torque sensor with your own equipment. Because of the wide variety of possible applications of the sensor, it is not possible to provide detailed instructions for your exact application. If additional information is needed, contact ***JR3***.

PHYSICAL MOUNTING OF SENSOR

CAUTION: THE SENSOR IS A PRECISION MEASURING DEVICE, AND MAY BE DAMAGED IF PROPER PRECAUTIONS ARE NOT OBSERVED WHEN IT IS MOUNTED.

Refer to the sensor specification sheet and drawing for detailed information about the rated capacity and mounting bolt pattern of your specific sensor. The sensor drawing also shows the positioning of the axes on the sensor. Install the sensor with the axes in the desired orientation. Use particular care when handling and installing sensors having low force ratings.

Most ***JR3*** sensors use captive button-head bolts to mount the sensor. The bolts are tightened with a hex key (provided with the sensor) through the bolt holes in the tool side of the sensor. DO NOT TURN ONE SENSOR BOLT ALL THE WAY IN AT A TIME. In many cases these bolts will protrude from the sensor even when fully retracted. If these bolts are tightened one at a time, the sensor can be damaged. Turn each bolt in a few turns at a time, then go to the next bolt until the sensor mounting surface is flat against your mounting surface. Then tighten the bolts in a sequence, moving from side to side. Torque these bolts in two or more stages until the torque recommended on the sensor interface drawing is reached.

BOLT TORQUE

The tables give torque values recommended by the bolt manufacturer to avoid damage to heads for the most commonly used bolt sizes. These values are for the captive button head screws used in standard sensors, other types of bolts may have different ratings. We recommend that sensor mounting bolts be torqued to the lower of the sensor's rated full scale torque, or to these values.

BE AWARE OF THE FORCE AND TORQUE RATINGS OF THE SENSOR WHEN TIGHTENING TOOL-SIDE BOLTS. Refer to the Sensor Load Ratings not the Electrical Load Settings. When tightening the mounting bolts on the tool side of the sensor forces and torques are exerted on the sensor. Be certain not to exceed the force or torque ratings of the sensor when tightening the mounting bolts on the tool side of the sensor.

English screws:

Size & Pitch	Recommended Tightening Torque
4-40	7 inlb
6-32	12 inlb
8-32	23 inlb
10-24 10-32	45 inlb
1/4-20 1/4-28	100 inlb
5/16-18 5/16-24	190 inlb (16ftlb)
3/8-16 3/8-24	300 inlb (25ftlb)
½-13 ½-20	1000 inlb (83ftlb)

Metric screws:

Size & Pitch	Recommended Tightening Torque
M3x.5	1.25 Nm (11 inlb)
M4x.7	2.9 Nm (25 inlb)
M5x.8	5.9 Nm (52 inlb)
M6x1.0	10 Nm (88 inlb)
M8x1.25	24 Nm (210 inlb)
M10x1.5	48 Nm (425 inlb)
M12x1.75	84 Nm (740 inlb)
M16x2.0	207 Nm (1800 inlb)

Fig. 2-1
Torque Values for *JR3* Sensor Mounting Screws

MOUNTING SURFACES

The sensor surfaces are precisely machined to be extremely flat. The plates or flanges to which the sensor is bolted should be as flat as possible, preferably within .0005" (half thousandth). Because of its compact size, the sensor is sensitive to the boundary conditions at the mounting surfaces. For this reason it is necessary that the mounting surfaces be stiff relative to the loads imposed. The mounting plate thicknesses shown in the table are recommendations for typical situations. Your particular application may require thicker mounting plates or be able to use thinner mounting plates.

Sensor Fx, Fy Load Rating in pounds	Aluminum Plate Thickness	Steel Plate Thickness
15	1/2"	3/8"
25	1/2"	3/8"
50	3/4"	1/2"
100	7/8"	5/8"
250	1 1/4"	7/8"
500	1 1/2"	1"

Fig 2-2
Recommended Mounting Plate Thickness

When mounting the sensor, or mounting a device to the sensor, be certain that no components are attached in such a way as to allow part of the load to pass around the sensor. No part attached to the "tool" or "load" side of the sensor should touch anything attached to the "robot" or "fixed" side of the sensor, or the sensor's protective canister, since it is attached to the "robot" or "fixed" side. If contact of this type is made, part of the load may be transmitted around the sensor resulting in false load readings.

There is a small gap between the protective canister and the face of the sensor on the tool, or load, side of the sensor. Be certain that no solid material or particles are allowed to lodge in the gap. If any solid is lodged in the gap, part of the load can be transmitted around the sensor resulting in false load readings. If the sensor environment

makes it likely that fluids or particles could enter the gap, a bellows or compliant booting is recommended.

CABLE ROUTING

Cable routing must allow for all possible movement of machinery. Do not allow stretching, crushing or excessive twisting of the cable. Where cable flexing is essential, it should be spread over a loop or length of cable rather than concentrated in a single spot.

CHAPTER 3
ELECTRICAL INTERFACE

Most sensors with internal electronic systems provide Digital output signals for use with several different ***JR3*** receiver systems. A few models are optionally available with Analog output.

DIGITAL OUTPUT

Digital output sensors transmit data to the receiver electronics in a synchronous serial format. All low level analog signals and the Analog to Digital (A/D) circuitry are within the sensor body, shielded from electromagnetic interference by the metallic sensor body. Data for all six axes is returned to the receiver at a rate of 8 kHz. Certain sensor models may utilize other data rates.

The data stream also includes feedback monitoring the sensor power supply voltage and information about sensor characteristics and calibration. Transmission of sensor calibration data from the sensor allows sensors to be interchanged with no need for any adjustment of the receiver circuitry. Feedback of the sensor power voltage allows use of long lengths of small gage wire in the sensor cable. Sensor power and data signals can be passed through slip rings with no increase in noise or loss of accuracy.

Standard digital output sensors utilize either a 6 pin RJ-11 or an 8 pin RJ-45 modular style jack depending on the sensor model. Other connectors may be used for special orders. See specifications provided with your sensor for details of connectors other than RJ-11 or RJ-45.

SENSOR CABLE

The cable for most ***JR3*** digital sensors is a flat modular type cable with RJ-11 (6 pin) or RJ-45 (8 pin) modular plugs. If your sensor does not utilize the flat modular type cable refer to information provided with the sensor for information about the cable.

JR3 digital sensors use a "standard" modular cable, as opposed to a "reverse" cable. The terms "standard" and "reverse" in regard to these cables are the cause of a great deal of confusion and misunderstanding. Therefore figure 3-1 showing the wire arrangement at the 2 ends of our normal 6 wire modular cable is provided to define our cable. The 8 pin cable is similar with the

addition of 2 wires. Either end of the cable can connect to the sensor or the receiver/processor. Note that there is a reversal of pin order between sensor and receiver/processor.

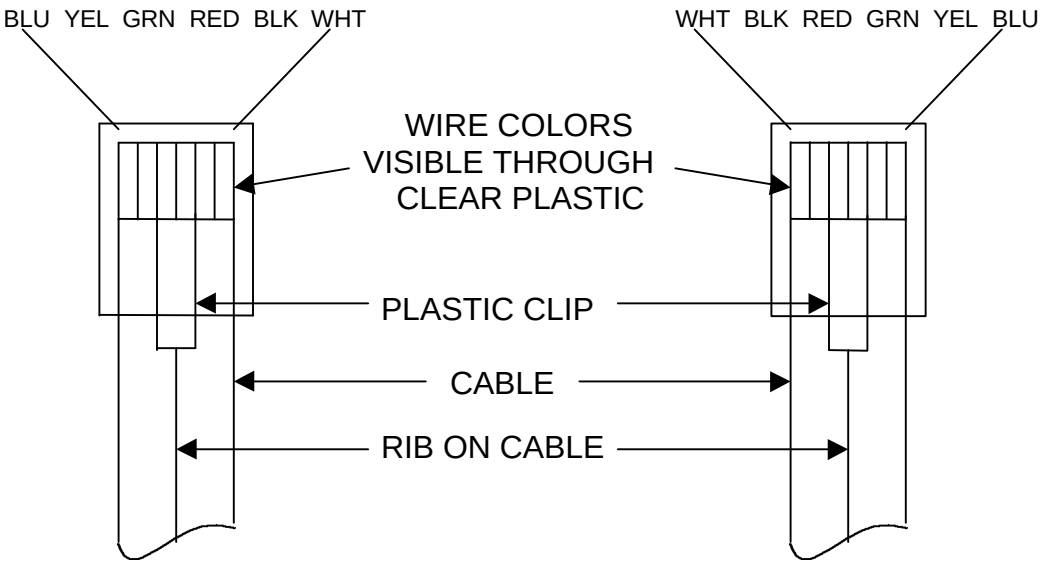
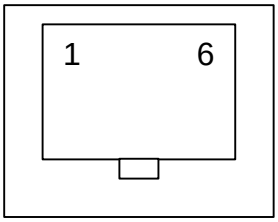
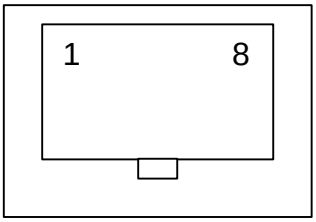


Fig 3-1
6 PIN SENSOR CABLE

The sensor uses either a 6 pin 8 pin or 10 pin modular jack depending on the model; receiver/processors use a 10 pin modular jack designed to allow connection of a 10, 8 or 6 pin plug with no damage. When a 6 pin plug is plugged into the 10 pin jack only the central 6 pins are used. There are 2 unused jack pins on each side of a 6 pin plug. Similarly, an 8 pin plug has 1 unused jack pin on either side.



6 pin Sensor Jack



8 pin Sensor Jack

PIN	SIGNAL
1	DCLK +
2	DCLK -
3	+ ~ 8 V
4	PWR COM
5	DATA +
6	DATA -

PIN	SIGNAL
1	- 12 V
2	DCLK +
3	DCLK -
4	+ ~ 8 V
5	PWR COM
6	DATA +
7	DATA -
8	+ 12 V

Fig 3-2
MODULAR SENSOR CONNECTOR PIN ASSIGNMENTS

Note that for the six central pins the function of each pin is identical though the pin numbers are different. This allows a sensor with six pins or a sensor with eight pins to connect to the same receiver electronic system jack.

ANALOG OUTPUT

Standard analog output sensors utilize a DE-9P connector. Other connectors may be used for special orders. See specifications provided with your sensor for voltage requirements and for details of connectors other than DE-9P.

PIN	SIGNAL
1	Fx
2	Fy
3	Fz
4	Mx
5	My
6	Mz
7	+ power
8	– power
9	power and signal common

Fig. 3-3
DE-9P SENSOR CONNECTOR PIN ASSIGNMENTS

Sensors with high level analog output provide an analog voltage for each axis, typically specified as +5 V to -5 V or +10 V to -10 V representing + full load to - full load. Data from analog sensors must be processed using the Calibration Matrix provided with the sensor. Refer to Chapter 4 for information about the decoupling matrix. When digital output sensors are used the receiver electronic system automatically processes the calibration matrix.

GROUNDING

As with any sensitive electronic system, improper grounding can cause problems. The receiver electronics and the equipment the sensor is mounted on should be connected to the same ground. In unusual situations, where there is a severe noise source near the sensor, it may be necessary to add a heavy ground connection between the sensor mounting surface and the receiver electronic system.

RECEIVER ELECTRONIC SYSTEM

Several different receiver/processors and electronic systems can be used with ***JR3*** sensors with serial data output. They receive the serial data from the sensor and provide sensor loading data in the desired form. Different electronic systems allow direct connection to computer and controller back planes, serial or parallel ports.

CHAPTER 4
CALIBRATION MATRIX

All multi-axis sensors have some degree of cross coupling, a condition where a force or moment load on one axis produces a change in the indicated load of other axes. Each ***JR3*** sensor is individually calibrated, with loads applied to each axis. The calibration data is used to generate a calibration and decoupling matrix, which is used to convert the output voltages to force and moment loading data in engineering units.

DIGITAL OUTPUT SENSORS

Sensor electronic systems with digital output store the matrix in non-volatile memory. When the sensor or electronic system is connected to the receiver the data is automatically downloaded in the first few seconds of operation. The receiver then applies the matrix to the sensor data stream without user intervention.

ANALOG OUTPUT SENSORS

When analog output electronic systems are used, the calibration matrix is provided on the Sensor Specification Sheet for your sensor. Sensors ordered with special calibration features may be provided with more than one matrix.

The six by six calibration matrix is multiplied by the six element voltage (column) vector. The result is calibrated force and moment data in the units specified on the Sensor Specification Sheet.

$$\begin{matrix} Fx \\ Fy \\ Fz \\ Mx \\ My \\ Mz \end{matrix} \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} & A_{1,5} & A_{1,6} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} & A_{2,5} & A_{2,6} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} & A_{3,5} & A_{3,6} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} & A_{4,5} & A_{4,6} \\ A_{5,1} & A_{5,2} & A_{5,3} & A_{5,4} & A_{5,5} & A_{5,6} \\ A_{6,1} & A_{6,2} & A_{6,3} & A_{6,4} & A_{6,5} & A_{6,6} \end{bmatrix} \times \begin{bmatrix} V_{FX} \\ V_{FY} \\ V_{FZ} \\ V_{MX} \\ V_{MY} \\ V_{MZ} \end{bmatrix} = \begin{bmatrix} Load_{FX} \\ Load_{FY} \\ Load_{FZ} \\ Load_{MX} \\ Load_{MY} \\ Load_{MZ} \end{bmatrix}$$

CALIBRATION MATRIX | OUTPUT VOLTAGE VECTOR | CALIBRATED LOADS IN ENGINEERING UNITS

Fig. 4-1
CALIBRATION MATRIX MULTIPLICATION

MANUAL CALCULATION:

Matrix calculations are most easily done with a computer, but they can, of course, be done manually. As an example, to find the decoupled, calibrated, Fx load reading using the raw analog voltages multilpy the Fx row ($A_{1,n}$) by the analog channel readings in volts. This gives the calibrated, decoupled, output in the units which were specified when the sensor was ordered.

$$(A_{1,1} \times V_{FX}) + (A_{1,2} \times V_{FY}) + (A_{1,3} \times V_{FZ}) + (A_{1,4} \times V_{MX}) +$$

$$(A_{1,5} \times V_{MY}) + (A_{1,6} \times V_{MZ}) = Load_{FX}$$

To find the loads for other axes, follow the same procedure using $A_{2,n}$ for Fy, $A_{3,n}$ for Fz, etc.

SAMPLE COMPUTER CODE FOR DECOUPLING:

A brief Pascal procedure showing one possible way of programming a computer for the decoupling matrix multiplication is shown below. This procedure can easily be implemented in other programming languages.

```
TYPE
   matrix = ARRAY[0..5,0..5] OF real;
   vector = ARRAY[0..5] OF real;

PROCEDURE mult (mat : matrix; vecIn : vector; VAR vecOut : vector);
VAR
  row : integer;
  col  : integer;

BEGIN
  FOR row := 0 TO 5 DO
    BEGIN
       vecOut [row] := 0;
       FOR col := 0 to 5 DO
          vecOut [row] := vecOut [row] + vecIn[col] * mat [row, col];
    END;
END;
```

Fig. 4-2
MATRIX SOFTWARE EXAMPLE

Anexo 4:

Hoja de características de tarjeta de 4 puertos PCI P/N 1593 de JR3 inc.

JR3, Inc. 22 Harter Ave, Woodland, CA 95776 (530) 661-3677

4 SENSOR PCI-BUS RECEIVER

RECEIVER OVERVIEW

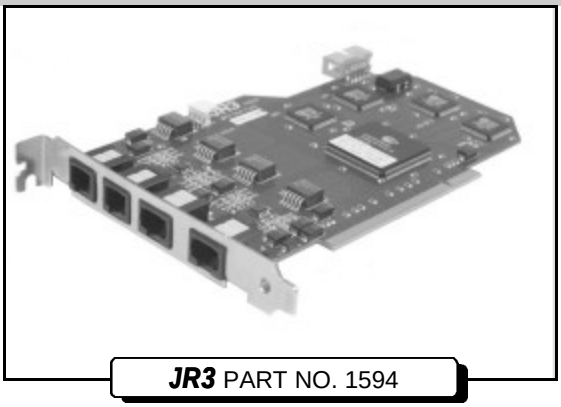
The 4 Sensor PCI-bus receiver is a member of the ***JR3*** family of serial receivers. It interfaces the PCI-bus to up to four of ***JR3***'s high speed, serial, six axis force and torque transducers simultaneously. It has a form factor which allows it to plug directly into any PCI bus slot.

The 4 Sensor PCI-bus receiver directly interfaces to the force sensors through small 6 or 8 wire cables. Two of the sensors must be 6 wire type sensors, the other two may be either 6 or 8 wire type sensors. The PCI-bus receiver uses the cable to provide power to the sensors, as well as to receive the high speed serial data from the sensors. The 4 Sensor PCI-bus receiver contains circuitry to monitor and adjust the power supply voltage to each sensor. The automatic power supply adjustment means that the sensor cable requirements are very forgiving. Long, small gage wires can be used with success. And since the PCI-bus receiver receives power directly from the computer bus, no external power supply is required.

The 4 Sensor PCI-bus receiver uses 4 Analog Devices ADSP-2184, 40 Mips digital signal processing chips. This chip has the ability to provide decoupled and filtered data at 8 kHz per axis. This data rate is an order of magnitude faster than previously available in the industry. Some of the signal processing functions performed by the PCI-bus receiver include: decoupling, coordinate transformation (translation and rotation), low-pass filtering, vector magnitude calculation, maximum and minimum peak capture, threshold monitoring, and rate calculations.

The 4 Sensor PCI-bus receiver communicates to the host computer through shared memory in the PCI bus address space. The interface adheres to PCI Local Bus Revision 2.2 Target Only specifications.

JR3 has serial sensor receivers available with a variety of interfaces. These include PCI-bus, ISA-bus, VMEbus, and Stäubli Unival. ***JR3*** also has many other interface options available. Please call our Applications Engineers to discuss your particular needs.



JR3 PART NO. 1594

Power Requirements:

The 4 Sensor PCI-bus receiver requires no external power. It draws power directly from the PCI bus and computer power supply. The receiver uses the following voltages and currents:

+5V - 870 mA typical
+12V - 25 mA typical (w/o sensor)
-12V - 5 mA typical (w/o sensor)

Each sensor will also draw anywhere from 200 to 400 mA from the +12V, and possibly as much as 100 mA from the -12V. If more than 2 sensors are connected, the Aux. Pwr. Connector must be connected to the computer power supply via a standard disk drive power cable.

Shared Data Space:

The ***JR3*** 4 Sensor PCI receiver's 16 bit wide DSP memory is mapped into the 32 bit wide PCI bus. The ***JR3*** PCI receiver is then simply accessed as memory on the PCI bus. Some of the data available and the addresses of that data are listed in the following table:

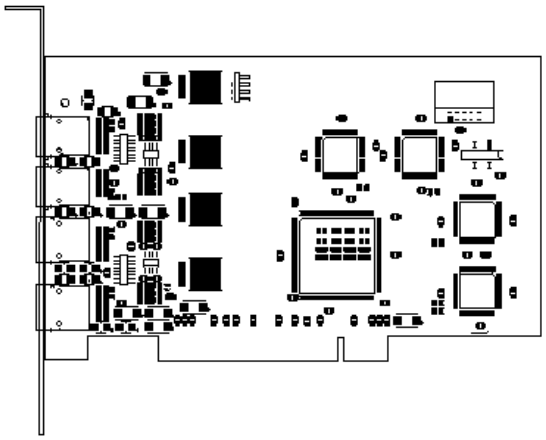
Addr	Data	Addr	Data
0x60	Shunts	0x68	Default Full Scale
0x70	Min Full Scale	0x78	Max Full Scale
0x80	Full Scale	0x88	Offsets
0x90	Filter 0	0x98	Filter 1
0xa0	Filter 2	0xa8	Filter 3
0xb0	Filter 4	0xb8	Filter 5
0xc0	Filter 6	0xc8	Rates
0xd0	Minimums	0xd8	Maximums

***JR3*, Inc. has been designing and manufacturing six axis loadcells since 1983. We have manufactured six axis loadcells ranging in diameter from 2 to 13 inches, and with load capacities from 2 to 25,000 lbs and 0.25 to 22,000 ft-lbs. Please feel free to call our applications engineers to discuss your particular needs.**

Address Selection:

The 4 Sensor PCI-bus receeiver occupies a 1024k-byte block in the PCI address space. The PCI bus is a plug and play bus, there is no user configuration necessary.

Outline of PCI-bus Receiver



Commands:

The PCI-bus receiver implements several commands which realize some of its more advanced features. Some of those commands, and their command numbers are:

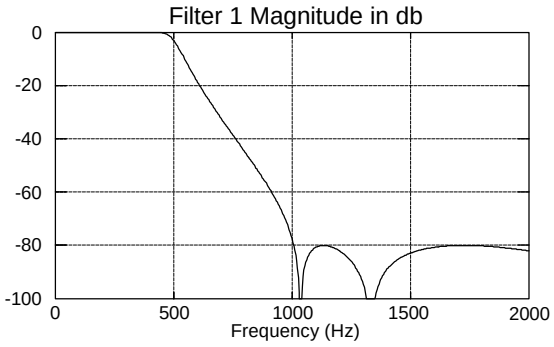
3:	Bit Set
4:	Bit Reset
5:	Use Coordinate Transform
6:	Use Stored Offset
7:	Set Offsets
8:	Reset Offsets
9:	Set Vector Axes
10:	Set Full Scales
11:	Read and Reset Peaks
12:	Read Peaks

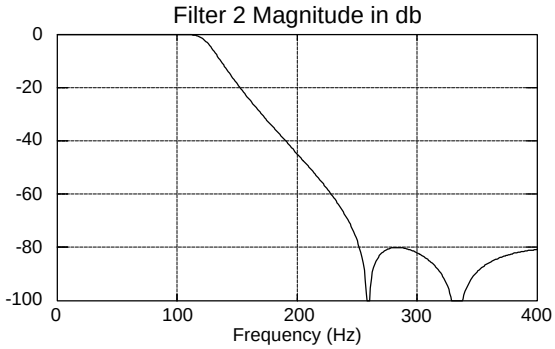
Coordinate Transforms:

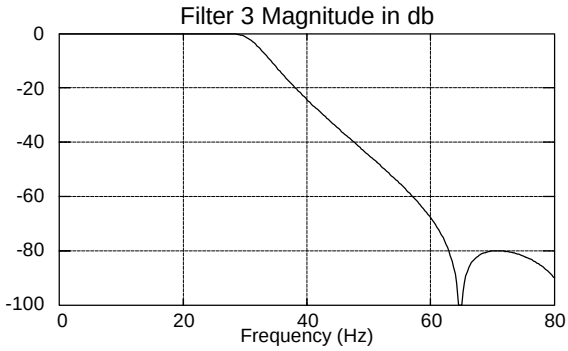
The PCI-bus receiver allows the force and moment data coordinate axes to be arbitrarily translated and rotated to any desired location and orientation. This allows the user to align the force and moment data with his coordinate axes, greatly simplifying data usage.

Digital Filters:

The PCI-bus receiver implements digital low-pass filters. Data for all 6 filters as well as unfiltered data is available at all times. These 6 filters have cut-off frequencies which are 1/4 of the preceding filter. The frequencies are ratioed from the sampling frequency of the sensor. For the typical 8 kHz sensor, the cutoff frequencies are 500, 125, 32, 8, 2 and 0.5 Hz.







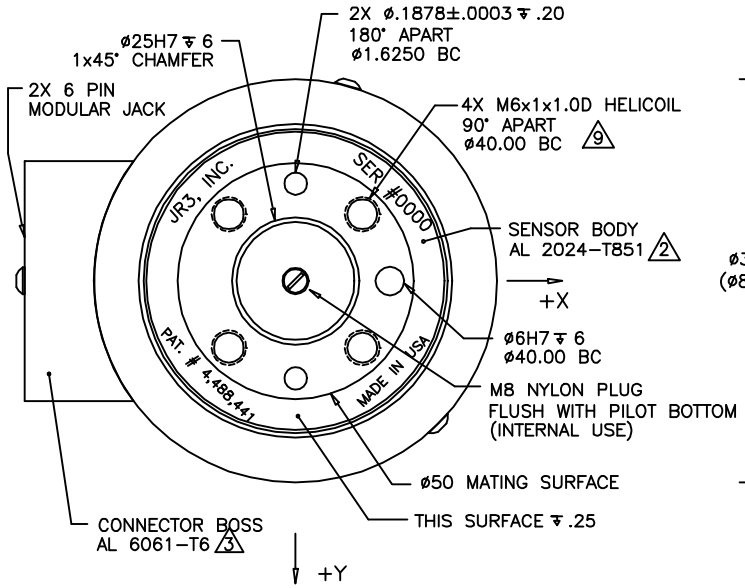
Vector Magnitudes:

The PCI-bus receiver implements two vector calculations for each set of data. These vectors can be calculated from any combination of force or moment data. Like the filter data, these numbers are available at all times.

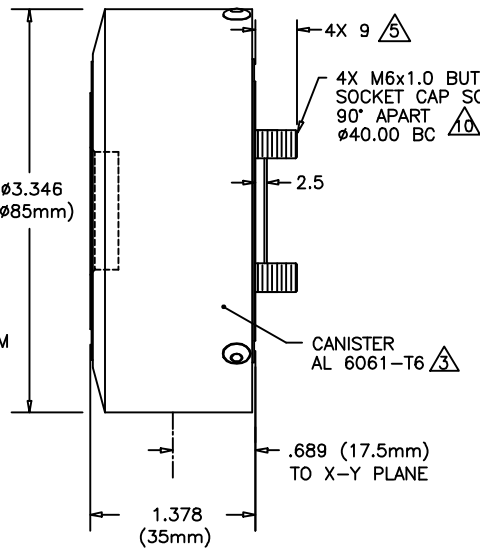
Anexo 5:

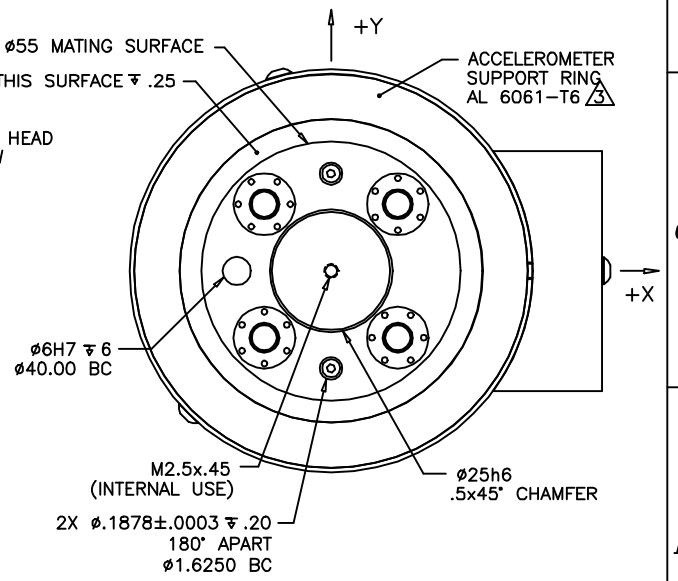
Plano de fabricación sensor fuerza/par 85M35A3-I40-DH12 de JR3 inc.

REVISIONS			
REV	DESCRIPTION	DATE	APPROVED
–	INITIAL RELEASE	10/31/01	G. SAKONA



TOOL VIEW





ROBOT VIEW

NOTES:

1. DIMENSIONS ARE IN INCHES UNLESS OTHERWISE NOTED.
2. TYPE 2, CLASS 2, BLACK ANODIZE.
3. TYPE 2, CLASS 2, BLUE ANODIZE.
4. AVAILABLE LOAD RATINGS INCLUDE: 15, 25, 50LB, WHERE:
   FULL SCALE Fx, Fy = SENSOR RATING
   FULL SCALE Fz = SENSOR RATING * 2
   FULL SCALE Mx, My, Mz = SENSOR RATING * NOMINAL DIAMETER (2.6in)
5. LENGTH OF CAPTIVE BOLT PROJECTION MAY BE ALTERED PER CUSTOMER REQUEST.
6. AXES SHOWN ASSUME OPPOSITE SIDE HELD FIXED.
7. ROBOT VIEW IS RIGHT HAND RULE (JR3 STANDARD). THIS FORCES TOOL VIEW TO BE LEFT HAND RULE.
8. TORQUE MOUNTING SCREWS TO 10Nm (88inlb) IN 3 EQUAL STEPS, FOLLOWING A 1–3–2–4 SEQUENCE.
9. MAXIMUM ALLOWABLE SCREW PENETRATION INTO SENSOR TOOL SIDE IS 8mm (.30in).
10. SCREWS ARE CAPTIVE, FREE TO ROTATE, AND ACCESSIBLE WITH HEX KEY THRU SENSOR TOOL SIDE.

TITLE: INTERFACE, 85M35 SENSOR, 12DOF, I40, DIGITAL

MAT'L: AS SHOWN	MODEL: 85M35A–I40	DRAWN: GCS
TOLERANCES: (UNLESS OTHERWISE NOTED) X = ±.25mm, .X = ±.10mm, .XX = ±.025mm, ANGLES = ±0.1°		SCALE: 1/1 AND NOTED
		DWG SIZE: B
		DWG NO: 7522 REV: –
JR3, INC. WOODLAND, CA 95776		SHEET 1 OF 1

AUTHORIZED COPY