

**Universidad ORT Uruguay**  
**Facultad de Ingeniería**

# **Implementación en código abierto de un bus de campo sobre Ethernet para integración de máquinas-herramienta.**

Entregado como requisito para la obtención del título de Ingeniero en Telecomunicaciones e  
Ingeniero en Electrónica.

**Fernando Mederos - 62327**  
**Daniel García – 137414**

**Tutor: André Fonseca**

**2014**

## Declaración de autoría

Nosotros, Fernando Mederos y Daniel García, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano.

Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el proyecto de fin de carrera;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Daniel García

5 de Marzo, 2014



Fernando Mederos

5 de Marzo, 2014

## ABSTRACT

Este trabajo describe la implementación de un bus de campo sobre Ethernet para aplicación al control en tiempo real de máquinas-herramienta con control numérico (CNC). Se pone particular énfasis en la viabilización del uso de esta tecnología en aplicaciones de bajo costo y pequeña escala integrando proyectos de software de código abierto y hardware de grado consumidor. Concluye con la construcción de una plataforma de ensayo con movimientos a fin de evaluar desempeño y oportunidades de desarrollo.

## Palabras Clave

Bus de Campo, fieldbus, Ethernet, CANopen, RTnet, Xenomai, XMOS, Linux, CanFestival, CNC, Máquinas Herramientas.

# INDICE

Declaración de autoría.....	2
ABSTRACT .....	3
Palabras Clave.....	4
INDICE .....	5
CAPITULO 1. Introducción .....	8
Objetivo.....	8
1.1 Reseña histórica de los buses de campo .....	8
1.2 Características de un bus de campo.....	9
1.2.1 Tiempo Real .....	10
1.2.2 Abstracción e interoperabilidad.....	11
1.3 Arquitectura propuesta de la plataforma de control .....	11
1.3.1 RTnet.....	12
1.3.2 CANopen.....	13
1.3.3 LinuxCNC .....	14
1.3.4 XMOS .....	14
1.4 Ensayo del bus: péndulo de Furuta.....	15
1.5 Modelo de transferencia tecnológica basada en licencias “abiertas”. .....	16
CAPITULO 2. Implementación .....	18
2.1 Fundamentos .....	18
2.2 Requerimientos.....	20
2.3 Ensayos .....	21
2.4 Estado tecnológico de los buses de campo.....	21
2.4.1 Comparativa de buses.....	22
2.4.2 Resumen del Estado tecnológico.....	25
2.5 CANopen/RTnet.....	27
2.5.1 Referencia de costos.....	27
2.5.2 Protocolo de red .....	28
2.5.3 Protocolo de comunicación .....	29
2.6 Arquitectura XMOS .....	29
2.7 LinuxCNC .....	30
CAPITULO 3. Maestro.....	32
3.1 Unión de los proyectos: Xenomai, RTnet, CanFestival, LinuxCNC. ....	32
3.1.1 RTnet.....	34
3.1.2 CanFestival.....	35

3.1.3	LinuxCNC .....	36
3.2	Implementación de interfaces y etapas de trabajo .....	37
3.2.1	Etapas de integración LinuxCNC y RTnet .....	37
3.2.2	Configuración RTnet.....	40
3.2.2.1	Cálculo de parámetros de temporización RTnet .....	41
3.2.2.1.1	Duración del <i>time slot</i> .....	42
3.2.2.1.2	Tiempo de ciclo de la trama TDMA: .....	44
3.2.3	Etapas de integración RTnet y CanFestival .....	45
3.2.4	Etapas de integración CanFestival y LinuxCNC .....	48
3.3	Escalabilidad de la solución Maestro/Controlador.....	51
3.4	Posibles Mejoras .....	52
CAPITULO 4.	Esclavo .....	53
4.1	Arquitectura XMOS:.....	53
4.2	Implementación del Bus de Campo CANopen/RTnet .....	54
4.2.1	Gestión de acceso y formato de trama RTnet:.....	55
4.2.2	Implementación del protocolo de red.....	56
4.2.2.1	Fundamentos del código.....	58
4.2.2.2	Organización de Núcleos .....	60
4.2.3	Implementación del protocolo de comunicaciones CANopen .....	62
4.2.3.1	Modificaciones efectuadas a la librería CANopen.....	62
4.2.3.2	Controlador de movimientos CANopen-DS402 .....	63
4.2.3.3	Organización de Núcleos .....	64
4.3	Posibles mejoras a la implementación de RTnet.....	67
4.4	Posibles mejoras a la implementación del Esclavo .....	67
CAPITULO 5.	Plataforma de Ensayos .....	69
5.1	Hardware .....	70
5.1.1	Construcción mecánica .....	71
5.1.2	Codificadores rotativos .....	71
5.1.3	Interfaz inalámbrico .....	71
5.1.4	Driver PWM.....	72
5.1.5	Motor.....	72
5.2	Software .....	73
CAPITULO 6.	Ensayos .....	74
6.1	Ensayos sobre RTnet.....	74
6.1.1	Latencia.....	74
6.1.2	Jitter.....	76

6.1.3	Tiempo de ciclo.....	76
6.2	Ensayos sobre CANopen/RTnet.....	77
CAPITULO 7. Conclusiones .....		82
7.1	Latencia y <i>jitter</i> .....	82
7.2	Período de refresco.....	82
7.3	Desempeño en la aplicación.....	83
7.4	Posibles optimizaciones .....	83
Referencias Bibliográficas. ....		85
ANEXO 1: Integración de Xenomai, RTNet y LinuxCNC.....		87
ANEXO 2: RTnet.....		88
ANEXO 3: CanFestival.....		94
ANEXO 4: LinuxCNC:.....		96
ANEXO 5: Hardware del Esclavo CANopen/RTnet .....		99
ANEXO 6: Hardware del Péndulo de Furuta:.....		100
ANEXO 7: Fuentes de Código:.....		101

# CAPITULO 1. Introducción

## Objetivo

En años recientes numerosas soluciones para buses de control de procesos industriales en tiempo real se han volcado a la utilización de Ethernet como medio físico para el intercambio de información entre un control central y dispositivos remotos. Generalmente estas soluciones han incorporado algún elemento propietario, dificultando el desarrollo de soluciones a medida para aplicaciones de pequeña escala o de bajo costo.

Este proyecto tiene como objetivo implementar una plataforma para el control distribuido de procesos en tiempo real, abierta y libre siguiendo la tendencia actual en dirección a Ethernet como medio de transporte, previendo su aplicación en máquinas-herramienta de Control Numérico Computarizado (CNC).

Ethernet ofrece una gran ventaja como medio para buses de tiempo real por sobre otros y es la disponibilidad de hardware estandarizado de bajo costo ya incluido en gran cantidad de plataformas PC estándar así como en *Single-Board-Computers*.

## 1.1 Reseña histórica de los buses de campo

Desde comienzos de la década de 1980, con la introducción de los PLC -con sensores y actuadores más inteligentes-, la automatización comenzó a acelerar su desarrollo. La cantidad creciente de dispositivos usados en muchas aplicaciones, comenzó poner en evidencia la necesidad de reducir y simplificar los cableados. Así también, la madurez de la industria de la microelectrónica permitió desarrollar protocolos de comunicación elaborados. Fue en este tiempo que el término *fieldbus* o bus de campo comenzó a utilizarse.

En primer lugar las compañías dedicadas al negocio de la automatización, comenzaron a realizar desarrollos propietarios de tecnologías de bus de campo. Estos desarrollos tenían los días contados ya que el volumen que cada fabricante podía producir y vender no era suficiente para sostener la investigación y el desarrollo de la tecnología.



Sumado a esto los clientes tenían a disposición varias soluciones de distintos *vendors* que supuestamente mejoraban sus procesos, pero primero debían ser convencidos de la validez de estos nuevos conceptos. Los clientes estaban acostumbrados a dispositivos de lazo de corriente o de entrada/salida digitales sencillos, que eran perfectamente compatibles entre distintos fabricantes. Por esto se resistían al cambio y la variedad de opciones nuevas y propietarias no hacía más que confundir. De forma que la industria acordó la promoción de sistemas de especificaciones abiertos para poder garantizar la compatibilidad entre dispositivos de distintos fabricantes, así comenzaron a surgir los primeros estándares de bus de campo para que los clientes recuperaran su libertad de elección. De todas formas el proceso de estandarización llevo tiempo y no estuvo libre de disputas.

En 1999 seis empresas líderes en el rubro como Fieldbus Foundation, Fisher Rosemount, ControlNet International, Rockwell Automation, Profibus User Organization y Siemens acordaron en un memorándum de entendimiento terminar sus disputas en lo referente a buses de campo. Como resultado surgió el estándar IEC 61158 que sentó las bases para la adopción generalizada de la tecnología de los buses de campo por parte de la industria.

## 1.2 Características de un bus de campo

Probablemente una buena forma de presentar un bus de campo sea en contraposición a una red local o LAN tradicional. En ambos casos se trata de un conjunto de dispositivos inteligentes interconectados entre sí para compartir información, pero ambos tienen directivas divergentes desde dos puntos de vista diferentes:

	LAN	Bus de Campo
Cantidad de Información a Intercambiar	Alta	Baja
Importancia de la predictibilidad temporal del intercambio	Baja	Alta

Tabla 1.1

Estas diferencias son tan determinantes como para motivar el desarrollo de formas particulares a los buses de campo para implementar su red de comunicaciones. Fruto de estas diferencias, los buses de campo han sido desarrollados con dos criterios de optimización básicos:

- Eficiencia en uso del medio. Dada la baja cantidad de información que un dispositivo debe compartir en un momento dado los paquetes de datos son cortos.
- Eficiencia en el diseño de los protocolos. Los dispositivos remotos en general no disponen de gran potencia de procesamiento, los protocolos por lo tanto deben ser sencillos de implementar y eficientes.

Junto con estos dos aspectos de optimización, los requerimientos de cada aplicación en cuanto a temporalidad, topología y costos han desembocado en diferencias fundamentales con las redes de área local LAN [1].

### **1.2.1 Tiempo Real**

Desde el punto de vista de su desempeño hay varios aspectos que un bus de campo moderno debe atender para lograr el objetivo de controlar múltiples dispositivos remotos:

- Predictibilidad temporal. El bus debe poder respetar una cadencia preestablecida de eventos.
- Cronometraje. Debe ser posible establecer el momento preciso en que un dispositivo remoto ejecutará una acción, de la misma forma que debe ser posible conocer en qué momento preciso fue tomada una medición.
- Determinismo temporal. El bus debe ofrecer precisión en la medición de tiempos y en la ejecución de acciones temporizadas.
- Priorización de tráfico. Se debe poder diferenciar entre distintos grados de sensibilidad temporal de los paquetes de datos.
- Garantización de responsividad. El sistema debe poder detectar y manejar situaciones de incumplimiento de tiempos críticos.
- Sincronización. Deben implementarse medios para la acción conjunta (sincronizada) de múltiples dispositivos remotos.

- Cuotificación del medio. Se debe poder imponer la asignación de espacio en el medio de comunicaciones entre los dispositivos integrantes del bus.

### **1.2.2 Abstracción e interoperabilidad**

El bus de campo debe ser implementado con un alto grado de abstracción de los dispositivos remotos y la información intercambiada. Esta es condición esencial para permitir la interoperabilidad entre los distintos dispositivos integrantes del bus, independientemente de fabricantes, tecnologías y procesos.

## **1.3 Arquitectura propuesta de la plataforma de control**

El bus de campo que se propone como solución en este proyecto utiliza el protocolo de arbitraje de red RTnet para el transporte sobre Ethernet de tramas en capas física y de enlace, con el protocolo de comunicaciones CANopen en capa de aplicación. Se ensayará el bus de campo utilizando LinuxCNC como software de control central. Un corte por capas mostrando los distintos integrantes de la implementación se muestra a continuación:


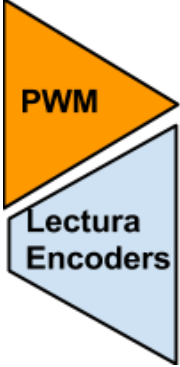






MAESTRO	Fieldbus	ESCLAVO
	Aplicación	
	Capa Aplicación	
	Capa Física y Enlace	
	RTOS	

Figura 1.1

### 1.3.1 RTnet

RTnet es un conjunto de protocolos para compartición del medio Ethernet desarrollado por el grupo de Sistemas de Tiempo Real de la Universidad de Hannover, Alemania. Define una estructura de acceso múltiple a un mismo medio de comunicación por medio de la división temporal (TDMA). Además implementa herramientas para el mantenimiento de una disciplina temporal determinística entre todos los nodos (dispositivos) participantes del bus independientemente de su ubicación y también para la configuración automática de éstos desde el nodo maestro.

Los lineamientos base del proyecto RTnet son [2]:

1. Lograr una solución para un protocolo de red en tiempo real, que se sitúe entre las soluciones fuertemente determinísticas implementadas en hardware y las soluciones de tiempo real “blandas” implementadas completamente en software con bajo nivel de determinismo.
2. Utilización de componentes populares, de forma de no depender de un proveedor en particular.

El proyecto mantiene un alto grado de abstracción de la información transportada y de los dispositivos intervinientes, sólo requiere para su implementación chips PHY/MAC estándar y sus restricciones de tamaño de trama son menores que los tradicionales de Ethernet. Ha sido desarrollado para un sistema GNU/Linux con *kernel* con capacidad *Real-Time* (RTAI o Xenomai), condición que le permite alcanzar un riguroso determinismo.

RTnet es software libre y de acuerdo con sus autores lo seguirá siendo. Los protocolos que lo conforman están completamente disponibles libres de licencias para ser utilizados en implementaciones. Además gracias al intercambio con otros proyectos industriales y de investigación durante su desarrollo, ha sido posible la interoperabilidad con diferentes paquetes de software. Factor que también ha ayudado al hecho de que ya existan drivers para las tarjetas de red (NIC) más populares en el mercado.

El punto fuerte de RTnet se aprecia cuando existe la necesidad de adaptabilidad a problemas y demandas específicas de un proyecto que haga uso de redes de comunicación de tiempo real. Los mecanismos de operación de los protocolos se exponen completamente posibilitando su customización y actualización, condición que hace posible la vigencia a largo plazo de las tecnologías desarrolladas a partir de éstos.

### **1.3.2 CANopen**

Los desarrolladores de RTnet han sugerido ya desde los inicios de ese proyecto la posibilidad de usar CANopen como capa de aplicación sobre RTnet para integrar un bus de campo [3]. Esta sugerencia parece totalmente sensata teniendo en cuenta que RTnet carece de las herramientas de alto nivel ofrecidas por un protocolo de comunicaciones, como son por ejemplo los perfiles de comunicación y las herramientas de gestión de red. CANopen

parece entonces un candidato obvio para complementar RTnet como protocolo de comunicaciones.

Nuestra investigación, nos encontró con el proyecto CanFestival desarrollado por Edouard Tisserant, Francis Dupin, Christian Fortin y Peter Christen. Es un proyecto con licenciamiento GPL y su objetivo es proveer un stack CANopen en ANSI-C independiente de la plataforma usada, ya sea PC, IPC de tiempo real o Microcontrolador. Implementa las especificaciones para capa de aplicación y perfil de comunicaciones de los estándares CiA DS301 y DS302 para componer un esclavo o maestro CANopen [4].

### **1.3.3 LinuxCNC**

Como software de control central se elige usar LinuxCNC, un paquete modular extremadamente flexible para el control de máquinas herramienta de todo tipo con licenciamiento GPL. Actualmente es desarrollado por un variado grupo de voluntarios y usuarios de equipamiento CNC sobre las bases del software de dominio público EMC, originado en el NIST en la década de 1980 y liberado al dominio público [5].

Este software incorpora un intérprete de Código-G para la programación de trayectorias de ejes coordinados. Esto se agrega a que está especialmente orientado al control de máquinas CNC, pero permite realizar tareas de control distribuido sobre cualquier clase de planta gracias a un módulo de abstracción de hardware (HAL) extremadamente poderoso.

### **1.3.4 XMOS**

Para la implementación del nodo esclavo, se optó por una arquitectura basada en un procesador XMOS multitarea de 16 núcleos lógicos. Este procesador ofrece rendimientos y facilidades comparables a las de una plataforma FPGA de rango medio. XMOS propone con sus procesadores una nueva aproximación a los sistemas embebidos. En lugar de poblar con periféricos específicos el entorno de un procesador central, se propone

programar los múltiples procesadores lógicos que integran el chip con la funcionalidad de cada periférico deseado [6].

XMOS pone a disposición un repositorio público de software licenciado para programar sus procesadores el cual abarca gran variedad de implementaciones de periféricos, entre los cuales cuenta un esclavo CANopen.

## 1.4 Ensayo del bus: péndulo de Furuta

Para los ensayos del bus implementado se construyó un péndulo de Furuta. Este péndulo consiste en un brazo horizontal giratorio motorizado y un segundo brazo giratorio pasivo. El objetivo del control realizado sobre el artefacto es regular la orientación del brazo pasivo mediante movimientos del brazo motorizado.

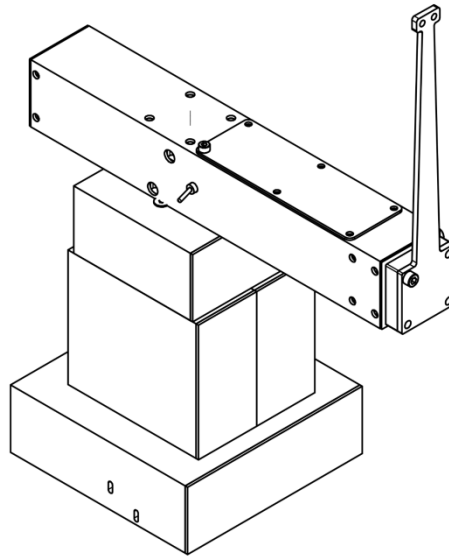


Figura.1.2: Péndulo de Furuta

## 1.5 Modelo de transferencia tecnológica basada en licencias “abiertas”.

Deseamos hacer énfasis en las motivaciones al uso de licencias “abiertas” antes de desarrollar los aspectos técnicos del proyecto en los siguientes capítulos. Entendemos que los aspectos legales y de licencias no son para pormenorizar en el desenvolvimiento de este u otros proyectos de desarrollo tecnológico dirigidos al contexto nacional o regional. El licenciamiento *Free Software* (hablaremos de este diciendo GPL, aunque hay variantes) u *Open Source* dando acceso a las fuentes del desarrollo permiten además de verificar su funcionamiento, **comprenderlo** y modificarlo en caso necesario de acuerdo con los requerimientos particulares del proyecto (teniendo los cuidados de uso y distribución de código que estos licenciamientos establecen, que no son iguales).

Además si se tomara la decisión de afrontar el desarrollo de un *stack* para comunicación en tiempo real y librerías para implementar protocolos de comunicación llevaría años contar con un marco de trabajo parecido al ofrecido actualmente por el conjunto RTnet/CanFestival en plataforma PC. Para viabilizar tal proyecto seguramente necesitaríamos contar con alguna industria con un interés específico que lo promocióne y apoye durante ese tiempo. Este escenario resulta improbable en el contexto de una universidad uruguaya.

Parece entonces importante tener en cuenta al menos para proyectos en esta área de aplicación, esta suerte de transferencia tecnológica y de conocimiento que hacen grupos de desarrolladores al resto del mundo. Hay que destacar que muy frecuentemente los propios desarrolladores involucrados en estos proyectos están disponibles por medio de *mailing lists* o foros, para responder en lo relativo a su desarrollo, implementación o funcionamiento.

Nos queda la convicción de que para desarrollar conocimiento y avanzar en la aplicación de tecnologías a la industria no debemos dejar de considerar desarrollos GPL u *Open Source*. Desearíamos que en Uruguay fuese más natural la búsqueda de buenas ideas plasmadas en proyectos de código abierto para desarrollarlas y utilizarlas. Probablemente una buena forma sea empleando desde los primeros años de estudio herramientas de código abierto o libre que nos acerquen la posibilidad de mostrarnos que también está a nuestro alcance participar del desarrollo tecnológico mundial.





## CAPITULO 2. Implementación

### 2.1 Fundamentos

Para acometer la integración de máquinas-herramienta el bus de campo debe resolver la comunicación entre el control central y los dispositivos remotos de entrega de potencia y toma de datos. Un esquema simplificado de interconexión entre los elementos participantes sin hacer uso de un bus de campo sería el siguiente:

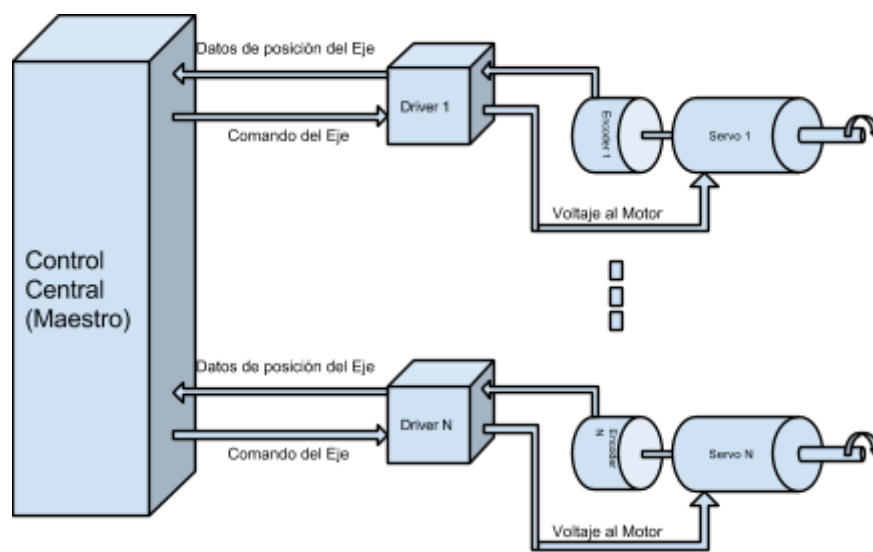


Figura 2.1

En este esquema los elementos que se desean controlar son servomotores destinados a dar movimiento a las distintas partes mecánicas de una máquina-herramienta. Estos servomotores son comandados por sendos Drivers encargados de entregar la potencia necesaria a los servomotores para su movimiento y también de tomar referencia de giro provista por Encoders (codificadores rotativos). La configuración en la cual se envían desde un control central señales de comando y se reciben datos de posición forma un **lazo de control** de posición o **lazo cerrado** de control de posición.

Cada Driver es comandado desde el control central de la máquina-herramienta. El control central es el encargado de coordinar los movimientos de los ejes simultáneamente respetando una secuencia programada. La secuencia puede requerir el movimiento compuesto por todos los ejes y debe ser mantenida a pesar de fluctuaciones en la carga recibida por cada uno de los ejes.

En la figura 2.1 se muestran 2 ejes de movimiento pero una máquina-herramienta típica requiere más que esto, generalmente por lo menos 3. Es frecuente que los datos de posición de cada eje provenientes de los codificadores no pasen por el driver sino que se encaminen directamente desde cada codificador hasta el control central. Si éste es el caso se denomina “lazo largo” al camino recorrido por la directiva de control, pero cuando el mismo Driver también utiliza el dato de posición (o velocidad) del codificador entonces aparece un nuevo lazo que se denomina “lazo corto”. El Driver puede utilizar el dato de posición para formar un lazo corto con objeto de mejorar la estabilidad del sistema, pero siempre es el control central quien ejerce el mando de la posición de cada eje.

Al aumentar la complejidad de la maquinaria, incrementar su tamaño y aumentar la cantidad de ejes a controlar se torna cada vez más complicado el manejo de una gran cantidad de señales a través de su estructura. Por cuestiones de eficiencia energética, limitación de la emisión de interferencias y aseguramiento de la integridad de señales usualmente se hace preferible la ubicación física de los Drivers en las inmediaciones de los servomotores. Aun así el tamaño de la máquina invariablemente incrementa el largo de cables de transporte de señales eléctricas hacia y desde el control central. Frecuentemente los cables para el transporte de señales de comando y datos de posición requieren una construcción especial que asegure la integridad de las señales, condición que encarece significativamente el cableado.

Atendiendo esta situación es que los fabricantes cada vez más están tendiendo a la aplicación de buses de campo para resolver el intercambio de información entre el control central y los Drivers desplegados por una máquina. Un esquema simplificado mostrando la aplicación de un bus de campo en la integración de una máquina-herramienta sería el siguiente:

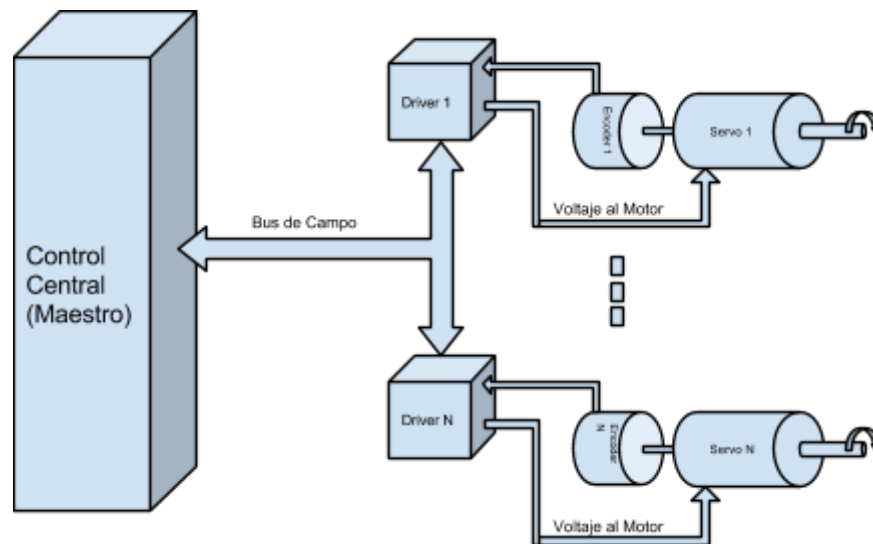


Figura 2.2

En la figura 2.2 el bus de campo es encargado del transporte de las señales de comando de posición hacia **todos** los Drivers y también de los datos de posición de **todos** los codificadores. Resultan evidentes en esta figura las ventajas potenciales de esta distribución para permitir la ubicación más apropiada del control central sin las restricciones impuestas por los cableados directos. Además de esto una construcción que haga uso de un bus de campo se puede tornar sumamente modular, facilitando el agregado y reposición de los componentes ya sean Drivers, servomotores o hasta el propio control central.

## 2.2 Requerimientos

Para acometer eficientemente su tarea, el control central deberá poseer información en tiempo real del movimiento de cada uno de los ejes y deberá instruir a cada driver también en tiempo real acerca de la potencia que debe entregar a su correspondiente servomotor. Resulta crucial entonces disponer de una comunicación rápida y confiable entre el control central y cada Driver.

Para dar solución a aplicaciones de pequeña escala o bajo costo es necesario dar énfasis a la reducción de costo de los dispositivos remotos, en particular previendo volúmenes de fabricación de decenas. Para ello se evitará que la implementación incurra en costos de licencias de software, hardware y royalties; también se buscará aprovechar avances en procesadores de bajo costo y alto desempeño evitando la incorporación de

chips FPGA en los dispositivos remotos. Por último pero no menos importante será objetivo realizar un trabajo en código abierto reutilizable para otros proyectos relacionados.

La finalidad prevista del bus de campo será integrar máquinas herramienta de uso industrial típicas como tornos, fresas y centros de mecanizado. Se considerarán prestaciones objetivo mínimas para el comando de 5 ejes con cambio automático de herramientas. Los diferentes tipos de dispositivos remotos previstos para formar parte de la máquina-herramienta son manejadores de motores (Drivers), variadores de frecuencia, puertos Entrada/Salida digitales y puertos Entrada/Salida analógicos. El bus debe ser capaz de cubrir los requerimientos de atender al menos 5 ejes realimentados en lazo largo y 64 puntos de entrada/salida digital con un tiempo de ciclo completo de 1ms como máximo.

## **2.3 Ensayos**

La plataforma de ensayos a utilizar para la evaluación y demostración del bus de campo será una implementación simple del péndulo de Furuta. Este péndulo es un sistema bastante apropiado al ambiente de laboratorio y requiere comandar un eje rotativo y tomar lectura de dos ejes rotativos. Los requerimientos de velocidad son comparables a los de la aplicación objetivo y si de futuro se deseara evaluar el desempeño con más ejes se podrán agregar más péndulos.

## **2.4 Estado tecnológico de los buses de campo**

Como se anticipó en la introducción progresivamente se está extendiendo la aplicación de Ethernet hacia ambientes industriales, haremos una revisión simplificada de algunas de las principales opciones disponibles de buses de campo utilizando capa física de Ethernet.

### 2.4.1 Comparativa de buses

Los aspectos tenidos en cuenta en esta comparación cualitativa son:

- Tolerancia a ruidos. El ambiente industrial es ruidoso por naturaleza, múltiples sistemas electrónicos comparten espacios restringidos y es frecuente el transporte de potencias considerables por conductores con los que el bus de campo debe coexistir. Estas situaciones tienen el potencial de afectar significativamente el desempeño del bus de campo.
- Redundancia de medio físico. Se comparan las capacidades de cada bus de campo para mantener operatividad aunque sea limitada en condiciones de caída de un enlace de la red.
- Flexibilidad de topologías. Se consideran las distintas alternativas de diseño del bus en cuanto a la organización de los enlaces considerando que la disposición física de los dispositivos remotos dependerá en gran medida de las particularidades de cada planta.
- Tolerancia a la pérdida del maestro y de dispositivos remotos. Se considera la capacidad del bus de mantener operatividad aunque sea limitada en condiciones de caída del maestro o de alguno de los dispositivos remotos. La eventual rotura o desconexión del maestro del bus puede ocasionar la pérdida total de la funcionalidad del bus si no está previsto un plan automático de contingencia.
- Tiempo de ciclo. Es la principal comparación de la velocidad del bus para reaccionar frente a los eventos en la planta. En general es muy dependiente de la cantidad de nodos (dispositivos) integrantes de la red y la cantidad de información que cada nodo intercambie con el maestro. Cada fabricante u organización de soporte especifica el desempeño de su bus bajo condiciones particulares por lo cual se imposibilita una comparación cuantitativa, limitamos esta comparación entonces a lo cualitativo en base a los requerimientos establecidos para el proyecto.
- Sincronización de relojes. Se evalúa la capacidad de cada bus de campo para mantener una referencia temporal entre todos los dispositivos remotos, condición necesaria para la sincronización precisa de eventos.

- Capacidad de compartir el bus con tráfico de baja prioridad (no-realtime). Frecuentemente es de gran utilidad reutilizar el bus para funciones de diagnóstico, mantenimiento o supervisión remotos de dispositivos o procesos, para ello generalmente no es requerida la rigurosidad temporal del bus pero sí la transferencia de volúmenes importantes de información como imágenes o archivos “log”.
- Herramientas para desarrollo y depuración. Condición fundamental para el desarrollo de un proyecto en código abierto será la disponibilidad de herramientas de acceso libre o la disposición de los fabricantes u organizaciones de soporte del bus de campo para ofrecerlas. Son de tener en cuenta los condicionamientos a veces impuestos desde los fabricantes u organizaciones para dar acceso a las herramientas, ejemplos, documentación, etc. que permitan llevar a cabo implementaciones.
- Hardware soportado. Se considera el nivel de soporte de hardware estandarizado para la implementación de nodos maestros y esclavos. Hardware propietario (royalties), costoso o de limitada penetración en el mercado dificultará la implementación.
- Licenciamiento. El presente proyecto ambiciona una implementación con licencia GPL, se evalúa la posibilidad de licenciar en este formato las implementaciones hechas a partir de ejemplos, librerías, drivers y aplicaciones aportadas por el fabricante u organizador de soporte.

	<b>EtherCAT</b>	<b>Powerlink</b>	<b>Sercos III</b>	<b>ModBus TCP</b>	<b>RTNet/CANopen</b>
<b>Tolerancia a Ruidos Electromagnéticos (por sobre lo estándar de la norma 802.3)</b>	Los datos para todos los nodos van en una única trama Ethernet -> Aumenta tamaño de trama al incrementarse número de nodos -> se incrementan los tiempos perdidos en retransmisiones frente a eventos de IEM	Tamaño de trama independiente de cantidad de nodos -> no aumenta el tiempo perdido en retransmisiones frente a eventos IEM al aumentar número de nodos.	Tamaño de trama independiente de cantidad de nodos.	Tamaño de trama independiente de cantidad de nodos.	Tamaño de trama independiente de cantidad de nodos.
<b>Redundancia de medio físico</b>	Implementa topología anillo. Recovery < 15µs	Implementa anillo.	Soporta topología anillo. Recovery < 25µs No estrella ni árbol	No	No

<b>Flexibilidad de topologías</b>	Anillo Línea	Todas las posibilidades con hubs/switches + Anillo Línea	Anillo Línea	(soporta hubs/switches) Estrella	(soporta hubs/switches) Estrella Daisy-Chain
<b>Tolerancia a pérdida de maestro y/o esclavos</b>	Soporta redundancia de Maestro	Soporta redundancia de Maestro	Sin redundancia de Maestro. Frente a caídas de 1 esclavo recupera en < 25µs	Sin redundancia de Maestro. No es afectado por caída de esclavos.	Maestro activo y maestros stanby. Recuperación "on-the-fly". No es afectado por caídas de Esclavos.
<b>Tiempo de ciclo dada una plataforma hardware</b>	<p>Ejemplos de tiempo de Ciclo según aplicación:</p> <ul style="list-style-type: none"> <li>• 256 I/O digitales en 11 µs</li> <li>• 1000 I/O digitales distribuidas en 100 nodos en 30 µs</li> <li>• 200 I/O analógicas(16 bit) en 50 µs, con 20 kHz de tasa de muestreo</li> <li>• 100 Servo-Axis (cada una con 8 Byte I+O) en 100 µs</li> <li>• 12000 I/O digitales en 350 µs</li> </ul> <p>Ver [7].</p>	<p>Depende de la implementación, como ejemplo de variación:</p> <p>1- Ejemplo de aplicación:</p> <ul style="list-style-type: none"> <li>• 6 Drives</li> <li>• 2 nodos I/O</li> <li>• 400 m de Cable</li> <li>• Cycletime: 291µs</li> </ul> <p>2-Ejemplo de aplicación:</p> <ul style="list-style-type: none"> <li>• 40 Drives</li> <li>• 50 nodos I/O</li> <li>• 500 m de Cable</li> <li>• Cycletime: 2347µs</li> </ul> <p>Ver [7].</p>	<p>Según [8] : Para sistemas Multi-axiales con procesamiento de señales centralizados el Cycletime está entre 31.25 µs a 125 µs</p> <p>Según [9]: Cycle time : 31,25 µs a 65 ms, no especifica la aplicación.</p>	ModBus TCP, va a tener los delays relativos a la red por usar TCP y usar un modelo maestro esclavo.	Depende de la implementación. Esquema de arbitraje similar a Powerlink, es esperable un desempeño similar. Se ensayará en este trabajo.
<b>Manejo de sincronización de relojes</b>	<p>Jitter &lt;1 µs</p> <p>La sincronización se implementa mediante un mecanismo llamado "Distributed Clock" (DC). Este mecanismo maneja una precisión de menos de 1 µs en la sincronización entre relojes.</p> <p>Ver [10].</p>	<p>Jitter &lt;1 µs</p> <p>Precision Time Protocol (PTP) IEEE 1588, Standard for a Precision Clock Synchronisation Protocol for Networked Measurement and Control Systems.</p> <p>Ver [11].</p>	<p>Jitter &lt; 1 µs</p> <p>Para sistemas Multi-axiales con procesamiento de señales centralizados, la precisión entre relojes para la sincronización es menor a 1 µs.</p> <p>Ver [8].</p>	<p>No sincroniza Relojes</p> <p>No maneja ningún mecanismo para sincronizar relojes, ModBus es un protocolo de capa de aplicación del modelo OSI.</p> <p>Ver [12]</p>	No especifica jitter. El protocolo de calibración de tiempo de transporte utiliza resolución de 1ns. Dependerá de la implementación.
<b>Compatibilidad con tráfico non-realtime</b>	Si, previsión de slot NRT (Non-RealTime) llamado Mailbox-Datagram	Si, previsión de slot asíncrono.	Si, implementa canal non-real-time.	Coexiste transparentement e con tráfico non-realtime	Si, previsión de puerto virtual asíncrono.



<b>Disponibilidad de herramientas de desarrollo/de puración</b>	Wireshark [13]  Herram. Desarrollo / librerías.	Filtro Powerlink para Wireshark [14]  Herram. Desarrollo / librerías.	Filtro Sercos III para Wireshark [15]  Herram. Desarrollo / librerías: OSADL (librería de drivers open-source) COSEMA (librería maestro open-source)"	Filtro Modbus TCP para Whireshark [16]  Herram. Desarrollo / librerías: FreeModBus libModBus MBServer"	Wireshark Plugin for RTnet [17]  Herram. Desarrollo / librerías: CANFestival (librería opersource) CANopenNode (GNU Software Stack)"
<b>Hardware Soportado</b>	Maestro: ubicuo  Esclavo: escaso (PHYs poco comunes)	User-Space: ubicuo  Kernel-Mode: Abundante KernelMode: Intel 82551ER 82562 82573L 82574 RealTek RTL8139 RTL8111B/C (only via pcap)	Muy escaso  Maestro requiere hardware especial (SERCOS Master Card). Existe Software-Master pero sólo soporta un IP-Core propietario.	Abundante  Hay tanto: hardware específico; como desarrollos para plataformas PC X86, SBCs y Micro-controladores.	Abundante  Intel 8255x EtherExpress Pro100Intel PRO/1000 (Gigabit Ethernet)DEC 21x4x TulipRealTek RTL8139RealTek RTL8169 (Gigabit Ethernet)AMD PCnet32/PCnetPCI VIA RhineNatSemi DP8381xMPC8xx (SCC and FEC Ethernet)MPC8260 (FCC Ethernet)MPC5200SMSC LAN91C111
<b>Licenciamiento</b>	Maestro libre (software), esclavo privativo (firmware/software)	Sin licencias ni patentes, abierto.	Maestro y esclavos licenciados.	Requiere licencia pero es gratis.	Sin licencias ni patentes, abierto.

Tabla 2.1 Comparación de buses de campo

## 2.4.2 Resumen del Estado tecnológico

En cuanto a flexibilidad de organización topológica del bus no hay una gran disparidad y la mayor diferencia entre buses radica en si soporta o no la topología de anillo. El único bus que **no** especifica la posibilidad de una organización en anillo es RTnet, aunque al igual que ocurre con el resto de los buses si se implementan nodos esclavos con hardware no-estándar sí se podría soportar esa topología.

En general los buses de campo considerados ofrecen un mismo nivel de tolerancia a interferencias electromagnéticas fruto de utilizar un mismo medio físico. Cabe señalar el caso de EtherCAT que incrementa la velocidad del bus condensando la información destinada a **todos** los dispositivos remotos en una misma trama Ethernet; esto trae una desventaja en caso de corrupción de trama para el bus de campo EtherCAT ya que todos los dispositivos remotos pierden información. Que tan significativo es este efecto depende en directa media de la relación S/N presente en el medio y la cantidad de dispositivos en la red.

Es general también entre los buses la capacidad de tolerancia a caídas del maestro y de esclavos. Caso especial es Sercos III que no especifica esta capacidad pero cabe suponer que sí la tiene. La posibilidad de reutilizar el bus de campo de tiempo real para aplicaciones menos demandantes, la capacidad de sincronización de relojes y la disponibilidad de herramientas de depuración son también características en común de la generalidad de los buses. La herramienta de depuración disponible para todos los buses de campo es *wireshark* con la adición de un *plugin* específico, una ventaja significativa del uso del medio Ethernet.

Al contrario de estas características ya mencionadas que resultan ubicuas hay 4 áreas en las que los buses ofrecen distintas aproximaciones y pueden derivar en decisiones importantes.

Una de estas áreas es el tiempo de ciclo, principal punto de comparación de la capacidad de los buses de transportar información rápidamente. La información ofrecida por los representantes de los buses a este respecto no permite una comparación sencilla ya que generalmente se presentan casos de aplicación en condiciones parcialmente descritas. De todas formas se puede inferir aproximadamente la capacidad esperable de un bus de campo acompasado con las tecnologías actuales y ésta cubre con facilidad el objetivo previamente planteado de actualizar 5 ejes y señales accesorias con una tasa de iteración de período máximo de 1ms.

Un segundo punto en el que existe diversidad entre las soluciones para buses de campo es el soporte de hardware. El peor extremo a este respecto es Sercos III que requiere de hardware propietario tanto para los nodos remotos como para el nodo maestro. Con mayor soporte se presenta EtherCAT caso en que el nodo maestro se puede implementar utilizando hardware estándar aunque no muy abundante ya que sólo algunos chips de capa

física Ethernet (chip PHY) son soportados. Aún así los nodos esclavos EtherCAT **requieren** hardware propietario dadas las características especiales de bus. Powerlink y RTnet en el otro extremo soportan gran cantidad de chips para acceso a capa física comunes en hardware de “*consumer-grade*”.

Las restantes dos áreas en las que conviene enfocar son la disponibilidad de herramientas de desarrollo y el licenciamiento. Como ya se mencionó estas áreas de comparación toman relevancia en nuestro proyecto dado el énfasis puesto al acceso de la tecnología a aplicaciones de bajo costo y/o pequeña escala. Comenzando por el caso menos favorable nuevamente está Sercos III con fuentes liberadas para una implementación de nodo maestro pero no ofreciendo documentación para la implementación de nodos esclavos más que a socios comerciales. Le sigue nuevamente EtherCAT que hace pública documentación y fuentes de ejemplo para la implementación de nodos maestro pero no para nodos esclavo, para nodos esclavo se deben firmar cláusulas “*non-disclosure*” que limitan la publicación del software que se desarrolle. Tanto Powerlink como RTnet ofrecen documentación y fuentes abiertas para todo el bus, cubriendo el nodo maestro y el esclavo.

## 2.5 CANopen/RTnet

Teniendo en consideración los objetivos de este proyecto toman especial relevancia los aspectos de disponibilidad de documentación y herramientas de desarrollo, licenciamiento y soporte de hardware por parte del bus de campo o sus patrocinantes. Es tan significativo el ahorro en costos de implementación que estos aspectos otorgan como para determinar la viabilidad o no de aplicar la tecnología de bus de campo a la integración de una máquina CNC de precio introductorio.

### 2.5.1 Referencia de costos

Componente (Omron)	Descripción	Precio (Digikey)
TJ1-STUDIO	Software de control de movimientos	US\$495

TJ2MC64	Unidad de control de movimientos	US\$2316
TJ2-ECT04	Manejador EtherCAT de 4 ejes	US\$1142
CJ1W-PA202	Fuente de poder	US\$190
R88D-KN01H-ECT-R	Driver servo 100W (se requieren x3)	3x US\$1914
	100m cable EtherCAT Phoenix Contact	US\$450
	<b>TOTAL</b>	<b>US\$10335</b>

Tabla 2.2

	Precio
<b>Router CNC 3 ejes 1.2m x 1.2m (fabricación China)</b>	<b>US\$6900</b>

Tabla 2.3

A modo de referencia en estos cuadros se muestra como empleando componentes para bus EtherCAT fabricados por la firma Omron apenas el costo de las partes destinadas a implementar el bus supera el de una máquina CNC completa de 3 ejes de construcción económica para el mecanizado de maderas (Router CNC de 3 ejes) [18].

## 2.5.2 Protocolo de red

RTnet es un paquete de software de código abierto y licenciamiento GPL para el arbitraje de una red Ethernet ofreciendo una comunicación de alta velocidad entre decenas de dispositivos remotos cada uno con acceso determinístico al medio. RTnet incluye fuentes y bibliotecas de drivers para arquitectura PC-Linux ya probados para placas Ethernet estándar (p.ej. Realtek 8169) a 100/1000Mbps y se encuentran en ensayo drivers para placas inalámbricas.

### 2.5.3 Protocolo de comunicación

El estándar CANopen de gran difusión en la industria ofrece una serie de perfiles normalizados para el intercambio de datos entre maestro y dispositivos remotos. Existen librerías de código abierto para la implementación de CANopen en arquitectura PC-Linux y XMOS.

En conjunto CANopen y RTnet permiten la implementación de un bus de campo con desempeño comparable a las opciones actualmente disponibles, pero de libre licenciamientos y hardware propietarios.

## 2.6 Arquitectura XMOS

Para la implementación de los dispositivos remotos optamos por una novedosa arquitectura de procesadores multitarea ofrecida por la firma XMOS. Esta arquitectura se presenta como una solución novedosa y atractiva con atributos de lógica programable, herramientas para la gestión de procesos en paralelo, alta potencia de procesamiento, bajo costo y extensas librerías de código abierto.

La arquitectura XMOS pone especial énfasis en un alto poder de procesamiento multitarea; de acuerdo con las especificaciones un procesador de rango medio como el XS1-L16A-128 es capaz de alcanzar 1000MIPS y distribuirlos equitativamente en hasta 16 núcleos lógicos ejecutando tareas concurrentemente.

Además de esto la arquitectura incorpora estructuras de hardware de interfaz que le permiten interactuar con señales externas de forma rápida y estable (mínimo *jitter*); por ejemplo *time-stamping* en hardware, serialización en hardware, la captura y generación de flancos pueden ser encargados al hardware sin recurrir a poleo obteniendo un desempeño altamente determinístico además de veloz.

La arquitectura incluye un sistema eficiente para la gestión de los ciclos de instrucción: cuando un núcleo entra en espera por un evento la ejecución es entregada a otros núcleos sin entrar nunca en bucles de espera ineficientes. Cuando el evento esperado

ocurre la ejecución es inmediatamente devuelta -con latencia máxima de 100ns- al núcleo que estaba esperándolo.

Fundamental para el aprovechamiento del procesamiento en paralelo es una estructura de comunicaciones rápida entre núcleos mediante canales unidireccionales que resuelve los problemas de sincronización de datos entre los núcleos lógicos. Estos canales de comunicación están implementados en hardware y además de comunicar núcleos de una misma pastilla pueden establecerse canales entre varias pastillas.

XMOS se ofrece como una arquitectura para soluciones embebidas pero sigue una filosofía diferente a la actualmente tradicional de incluir múltiples componentes de hardware embebidos en cada chip como ser UARTs, interfaces SPI, temporizadores, generadores PWM, etc. Por el contrario la filosofía es programar los componentes de hardware que se requieran escribiendo código para ser ejecutado en núcleos independientes. Esta solución para la implementación de hardware embebido resulta sumamente flexible; XMOS ya ofrece amplias bibliotecas de código abierto para la implementación de componentes estándar como UART, SPI, PWM, timers, etc. Cada núcleo dispone de estructuras de entrada/salida que permiten resolver todas estas funciones con el mismo desempeño de un módulo de hardware dedicado pero teniendo una gran ventaja en cuanto a flexibilidad de los componentes (alterando su código) y también customización del procesador (eligiendo los componentes deseados).

## **2.7 LinuxCNC**

La aplicación encargada de ejercer el comando en última instancia de la máquina-herramienta no es considerable como parte del bus de campo y por lo tanto no es estrictamente parte de la implementación del bus. Aún así es parte fundamental del conjunto que hace operativa a la máquina-herramienta y por tanto corresponde la elección de un software destinado esta tarea si se pretende un ensayo funcional del bus de campo.

La elección del software LinuxCNC como aplicación de control central obedece principalmente a que es un paquete modular y extremadamente flexible para el control de máquinas herramienta de todo tipo en tiempo real, incorpora un intérprete de Código-G

para la programación de trayectorias de ejes coordinados y además está disponible en código abierto con licencia GPL.

## CAPITULO 3. Maestro

Para la implementación del maestro, se mantuvieron las características que deben tener el *fieldbus* y la aplicación de usuario. El *fieldbus* debe ofrecer: determinismo temporal en su capas bajas (física y enlace); con una capa de aplicación: abstracta, flexible según el perfil de comunicación que exija la aplicación y que garantice interoperabilidad entre distintos dispositivos. La aplicación de usuario deberá ser en software, modular y que también aporte a la interoperabilidad. En todos los casos deben tener la posibilidad de funcionar sobre un *real time operating system* (RTOS), respetando su arbitraje temporal para compartir los recursos del sistema.

Así se tuvo en cuenta como punto de partida las librerías de RTnet y CanFestival que fueron elegidas para conformar el *fieldbus* del lado maestro. Estas están implementadas: en el primer caso como parte del proyecto Xenomai sistema operativos de tiempo real (RTOS) basados en un *kernel* GNU/Linux; y en el segundo caso tiene la posibilidad de ser compilado e instalado sobre sistemas Win32 o GNU/Linux, en particular con disciplinas de temporización para las tareas o *threads* controlados por el *kernel* RTOS Xenomai.

Sumado a lo anterior, como aplicación de usuario se decidió usar el LinuxCNC en su versión *real time* sobre Xenomai. Es una aplicación pensada para el control de máquinas CNC, que hasta la fecha de este trabajo no tiene implementado una interfaz para comunicarse sobre un *fieldbus* CANopen a un dispositivo que funcione como esclavo.

Así estos proyectos terminan en confluir en una misma plataforma de trabajo de tiempo real, pero carecen al momento de la posibilidad de integrarlos.

### 3.1 Unión de los proyectos: Xenomai, RTnet, CanFestival, LinuxCNC.

El desafío, del lado del maestro o controlador, surge en poder confeccionar las interfaces entre estas librerías, para poder desde el LinuxCNC, mandar los datos de control



a un esclavo, usando el *stack* CANopen disponible a través de CanFestival y transportarlo vía RTnet como capas física y de enlace.

Nota: Listamos las referencias para estos proyectos ya que solo entraremos en detalles relativos a su integración, no a su funcionamiento general:

- Xenomai [19]
- RTnet [20]
- CanFestival [4]
- LinuxCNC [21]

Para lograr una idea de cómo se logró entrelazar estos desarrollos acudimos a un diagrama de bloques y capas para aclarar la arquitectura pensada:

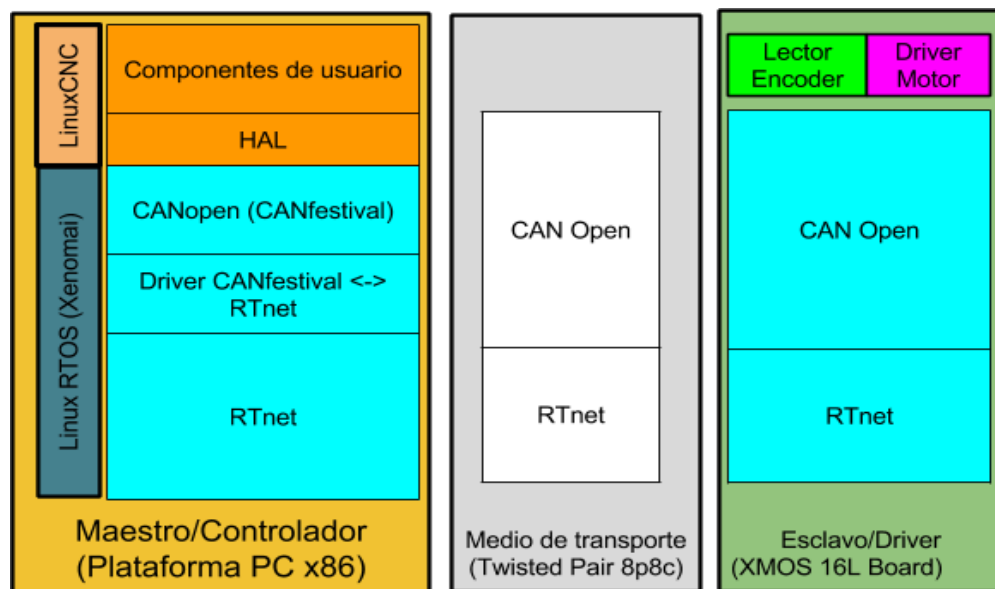


Figura 3.1

En la figura 3.1, se dispone un diagrama de tres bloques de la arquitectura del total del sistema que ocupa a este proyecto. Estos bloques son para el Controlador o Maestro, donde tenemos 2 diagramas de capas paralelos, uno ubicando el SO y aplicación, y otro de capas de los elementos del *stack* de comunicación. Luego el bloque central representa el medio de transporte, con las capas del *stack* de comunicación que conlleva. Y por último la arquitectura del esclavo XMOS será discutida en el capítulo 4.

En lo que ocupa al Control o Maestro, la arquitectura se basa en una plataforma PC x86, con una distribución Ubuntu, que corre sobre el *kernel* Xenomai. A la izquierda se muestra, dentro del bloque Maestro/Controlador, un diagrama de dos capas, una para el sistema operativo GNU/Linux-Xenomai y otra para la aplicación LinuxCNC. Luego a su derecha dentro del mismo bloque, se muestra que posición ocupa cada librería.

### 3.1.1 RTnet

El RTnet está encargado de manejar las comunicaciones de la NIC Ethernet en tiempo real. Dentro de las librerías RTnet, se encuentran un número de drivers codificados con instrucciones del API Xenomai/RTAI, particularmente de su librería RTDM, lo cual permite garantizar un acceso al medio con las restricciones que implementa este RTOS [22].

Entonces el RTnet, envía y recibe tramas Ethernet desde el NIC, usando una disciplina TDMA que funciona en la capa llamada RTmac. Una diferencia importante para el tiempo real es evitar el mecanismo para manejar colisión de tramas CSMA (*Carrier Sense Multiple Access*) que implementa el Ethernet IEEE.802.3, el cual introduce incertidumbre en determinar el momento de re-envío de una trama (también se deben evitar en los dispositivos de interconexión mecanismos como spanning tree u otra facilidad a nivel de capa MAC). Al usar un mecanismo TDMA, el acceso al medio es ordenado y la comunicación entre el maestro y un esclavo se realizará en un *time slot* predefinido, con un ciclo de trabajo también configurado de antemano, esto brinda determinismo en la comunicación de tramas, el momento de transmisión y recepción de datos es conocido siempre, no importa la cantidad de nodos que quieran compartir el bus.

Una vez configurado el RTnet, mediante sus archivos de configuración **rtnet.conf** y **tdma.conf**, entonces se integra al sistema operativo como módulos de *kernel* y el driver de tiempo real pueden enviar y recibir tramas Ethernet de forma determinística.

En nuestra configuración, el RTnet resuelve las capas 1 y 2 del modelo OSI, para llegar a la capa de aplicación sería necesario implementar el resto de las capas. Las capas 3 de red, 4 de transporte, 5 de sesión y 6 de presentación no son necesarias, por lo cual no se implementan. Esta decisión de implementación fue tomada debido a que el campo de

aplicación que tiene como objetivo este proyecto, se encuentra con topologías de red LAN de un solo segmento, la cual es totalmente compatible con una comunicación en bus, y no necesita de las funcionalidades que brindan estas capas. La disciplina TDMA del RTnet resuelve estas necesidades y no es necesario agregar otra capa previa a la de aplicación.

Alcanza usar un *stack* que resuelva la capa alta del modelo OSI, la de aplicación.

### 3.1.2 CanFestival

CANopen es un *stack* ampliamente usado, con la ventaja de tener sus especificaciones abiertas y disponibles. Contamos con el proyecto CanFestival, donde se brinda una API para integrarla a cualquier medio de comunicación realizando la implementación de un driver específico.

Para incorporar el stack CanFestival a nuestra arquitectura, fue necesario desarrollar un driver que permita enviar las tramas CANopen sobre RTnet. La estructura de esta librerías tiene previsto este mecanismo, donde respetando las funciones implementadas para ser usadas por el resto de la librería, se realizó el driver usando las llamadas a la API de RTnet [23]. En este paso, se implementó un tipo de trama Ethernet distinta, que llamamos RTCANopen con el código de *Ether type* = 0x9023. El driver **can\_rtnet**, arma las tramas Ethernet mediante un *socket RAW* [24], lo que nos permite minimizar la cabecera de la trama. Esta es armada por la función de envío de tramas de nuestro driver y no por las funciones del *kernel* como es en el caso de un socket TCP o UDP.

Las librerías de CanFestival tienen previsto, como ya dijimos, ser compiladas para el *kernel* Xenomai y enlazar el resto de las librerías al driver que se elija en la compilación (*configure*).

Una vez integrado el CanFestival al RTnet, ambos funcionando con instrucciones del *kernel* Xenomai, solo resta generar los datos a nivel de aplicación y enviarlos usando el API CanFestival.

### 3.1.3 LinuxCNC

La elección del LinuxCNC como aplicación de usuario, reúne toda la potencialidad de esta aplicación para controlar máquinas CNC, como también permite usar su HAL (*Hardware Abstraction Layer*) para realizar otras aplicaciones, como puede ser el control de una planta cualquiera [25]. La lógica del HAL, permite confeccionar componentes, tal como si fuera un hardware, e interconectar sus entradas y salidas de forma de obtener los resultados deseados.

El modelo más usado para la conexión a un dispositivo externo, dentro de las aplicaciones donde se usa LinuxCNC, es el de un componente driver para una tarjeta PCI específica que se comunica directamente con el *driver/encoder* del motor. Sin hacer uso de un protocolo de alto nivel como CANopen, y mucho menos transportar sus tramas sobre una interfaz Ethernet, se manejan las señales del driver directamente.

Entonces para integrar el CanFestival dentro del HAL de LinuxCNC, se decidió implementar un componente que funcione como un maestro CANopen. Donde se establece los valores de configuración para la comunicación vía *Process Data Objects* (PDO) y cuales *Communication Object IDs* (COBID) del diccionario de objetos serán mapeados a estos al cargar el componente, como también los *node-ID* del maestro y esclavo [26].

De esta forma, el usuario puede tanto usar los componentes que brinda el LinuxCNC, o programar uno propio para controlar vía CANopen un dispositivo que tenga implementado el RTCANopen como esclavo. En caso de ser necesario usar un medio distinto al RTnet, el mismo componente se puede compilar enlazando una librería de un driver distinto, por ejemplo CAN sobre una interfaz serial. Esto se hace cambiando el valor de una variable en el componente. Es una flexibilidad que presenta este desarrollo.

Lo expuesto anteriormente explica en suma, cómo se unió cada bloque de la implementación a grandes rasgos. Para profundizar en la implementación de estos elementos, en lo siguiente explicaremos, las etapas de trabajo para el desarrollo y detallaremos cada elemento e interfaz.

## 3.2 Implementación de interfaces y etapas de trabajo

Para obtener un sistema RTOS GNU/Linux, se decidió instalar un sistema con un *kernel* Xenomai. Así, como también dejar disponibles las librerías y APIs Xenomai, para construir sobre estas los desarrollos necesarios, se debe instalar su código fuente. Ahora para ser soportado por todas las librerías y aplicaciones necesarias, se eligió una distribución particular del Ubuntu 10.04 “Lucid Lynx”, donde viene integrada la versión 3.5.7 del LinuxCNC para tiempo real, con un *kernel* por defecto RTAI (versión 2.6.32-122). Esta es la última distribución Ubuntu LTS disponible al momento de escribir este documento con soporte de LinuxCNC.

Luego a esta instalación se instalan los paquetes con los parches del *kernel* para el “*upgrade*” a Xenomai versión 2.6.2.1, que es la versión soportada tanto por el LinuxCNC versión 3.5.7 como el RTnet versión 0.9.13.

### 3.2.1 Etapa de integración LinuxCNC y RTnet

Como primer hito del proyecto se desarrolló un componente para el HAL del LinuxCNC que soporte el envío de tramas sobre la interfaz RTnet generadas desde esta aplicación. Este componente fue llamado **prueba\_rt.comp**. Lo cual, usando el API de RTnet, nos permitió probar las distintas posibilidades del envío de tramas Ethernet, usando los distintos tipos de *sockets* que se pueden implementar. Las funciones de este componente son ejecutadas dentro de un *thread real time* creado y lanzado por el HAL, que hace uso de las librerías de Xenomai para el manejo de tareas. Es importante tener en cuenta que este componente es de tiempo real, y el HAL lo lanza como módulo del *kernel*, lo que contribuye a que las tareas/hebras que son lanzadas tengan una prioridad mayor dentro Ipipeline o Adeos del Xenomai, que aplicaciones en espacio de usuario [27].

En su realización se tomaron algunas decisiones de diseño que explicaremos. Para ello, primero es necesario explicar el *stack* RTnet. Este *stack* y sus interfaces están brevemente mostrados en el siguiente esquema [3]:

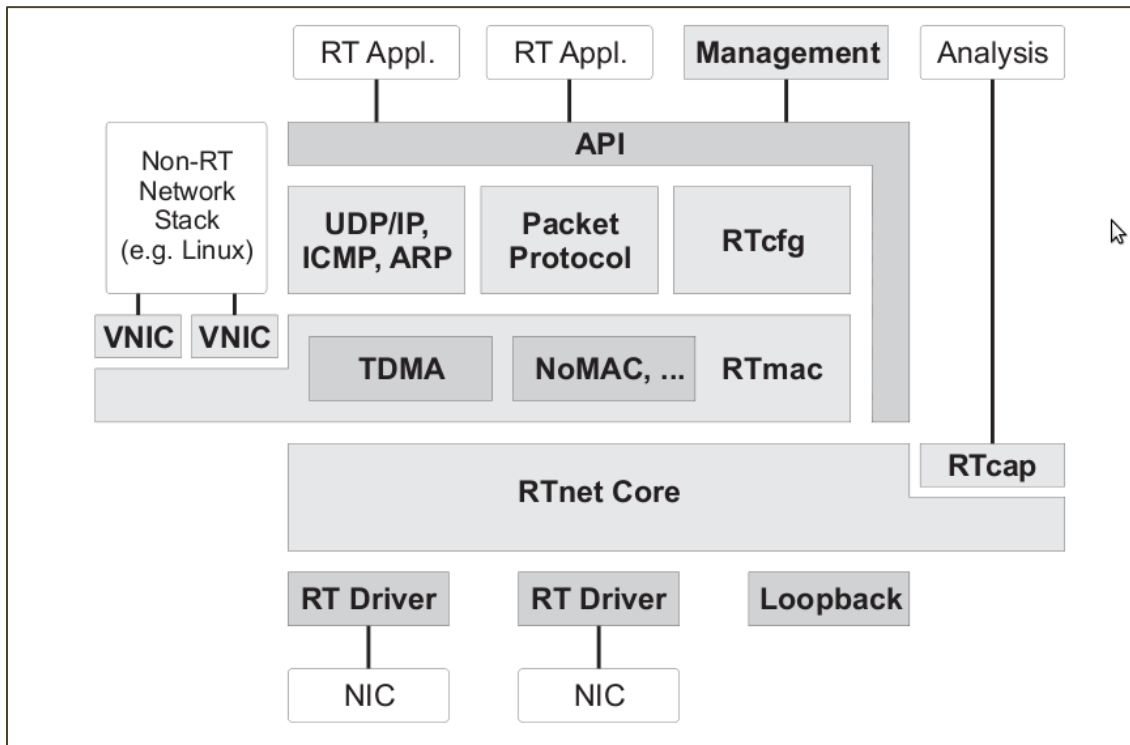


Figura 3.2

De estas capas y componentes solo se configuran e instalan:

- RT Driver (driver de tiempo real para la tarjeta Realtek 8169 y el Loopback)
- RTnet Core con RTcap para poder capturar y analizar paquetes
- RTmac (TDMA)

El resto de las capas no se entendieron convenientes ya que agregan *overhead* y señalización que no aporta ningún beneficio a nuestro objetivo. Lo que descartamos entonces es:

- RTcfg: sirve para configurar atributos RTnet automáticamente en dispositivos. El implementarlo agrega trabajo a ser incorporado del lado de los esclavos. En una aplicación de una máquina CNC los dispositivos estarán fijos y su configuración estática, por tanto no se implementó por entender que puede ser incorporado en un futuro trabajo.
- Los protocolos para transmisión y enrutamiento de paquetes, no son necesarios y agregan complejidad y demora en la comunicación. Las aplicaciones en la cual la comunicación se realiza como un *fieldbus* tienen una topología de red LAN de un segmento, el direccionamiento IP no es necesario. En caso de necesitar una

comunicación IP se puede usar la interfaz *Virtual Network Interface Card* que funciona en modalidad no tiempo-real.

Otro fundamento para eliminar estas capas, es que un paquete IP/UDP conlleva un *overhead*, que ocupa ancho de banda y tiempo de procesamiento. Para aplicaciones de tiempo real es importante cuidar cualquier demora que se pueda ahorrar en la comunicación. Dado que las fuentes de retardo y *jitter* ocurrirán principalmente a nivel de aplicación de usuario y latencia del sistema.

La holgura en la potencia de procesamiento de un sistema PC x86 o un microcontrolador/microprocesador es limitada. Nuestro objetivo es usar tecnología de grado consumidor, para que las aplicaciones de este proyecto puedan ser incorporados en emprendimientos de pequeña escala.

Entonces, como dijimos, para ahorrar *overhead*, se decidió implementar la comunicación vía un *socket RAW*. Donde la trama Ethernet no va a tener el encabezado de la IEEE 802.3, va a ser una trama Ethernet II. Se muestra en la siguiente figura:

Preámbulo	Dirección Destino	Dirección Origen	Tipo	Datos	CRC
8 Bytes	6 Bytes	6 Bytes	0x9023 (2 Bytes)	46 Bytes	4 Bytes

Figura 3.3

Sumado a esta diferencia el campo Datos se fija en 46 bytes (fijado por el PHY del esclavo).

En suma para dar un sentido gráfico a lo dicho antes, la implementación realizada para el RTnet tiene como resultante el siguiente esquema:

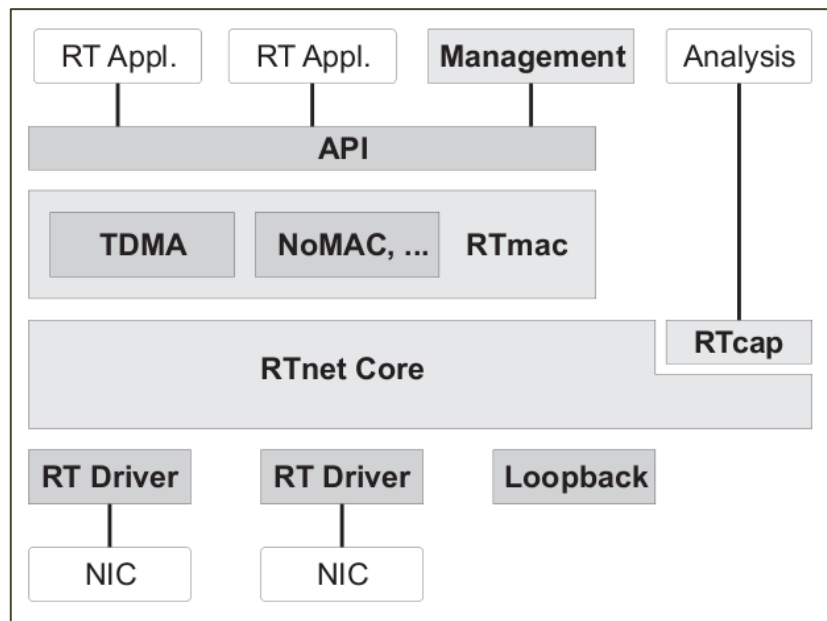


Figura 3.4

Donde el RT Driver que usamos es el `rt_r8169` para la placa de red con chip Realtek 8169. La RT Application va a ser el driver `can_rtnet` desarrollado para la librería CanFestival, que discutiremos luego.

### 3.2.2 Configuración RTnet

Los pasos de instalación del RTnet para configurarlo de esta forma se encuentran en los Anexos 1 y 2. Esta librería en conjunto con el Xenomai son el marco de trabajo *real time* que tendrá tanto el CANopen como el LinuxCNC. El RTnet como ya dijimos nos permite manejar el medio Ethernet, que en sí es agnóstico de determinismo. El Xenomai realiza las funciones de un RTOS para el manejo de tareas o *threads* para compartir los recursos de hardware de forma de garantizar arbitraje en la duración de ejecución y previsibilidad en la entrega de sus resultados.

Entonces, el Xenomai queda instalado y configurado una vez hecho el *upgrade* del *kernel*. En cambio el RTnet, además de instalar y compilar, es necesario configurar su funcionamiento con algunos parámetros. En lo siguiente haremos un cálculo de dichos parámetros teniendo en cuenta el sistema sobre el que va a funcionar.



### 3.2.2.1 Cálculo de parámetros de temporización RTnet

Ya se explicó a nivel módulos y capas del RTnet como fue pensada nuestra implementación, ahora daremos algunos detalles de la configuración que son relevantes.

Para configurar el RTnet se disponen de dos archivos de configuración, donde se asignan los parámetros de configuración: tipo de nodo, IP, máscara de red, RT driver a usar, etc. Dentro de esos parámetros, se encuentran el TDMA\_CYCLE y el TDMA\_OFFSET:

```
# Simple setup: Cycle time in microsecond
TDMA_CYCLE="1000"

# Simple setup: Offset in microsecond between TDMA slots
TDMA_OFFSET="50"
```

Estos parámetros se encuentran en el archivo rtnet.conf, el primero determina cuánto dura la trama TDMA y el segundo la duración de la ranura de tiempo o *time slot*. Para configurar estos valores conviene hacer algún estimado de que valores sería conveniente asignarles.

### 3.2.2.1.1 Duración del *time slot*

Nuestro punto de partida es que tenemos una interfaz que funciona a 100Mbps, lo llevamos a microsegundos que es la unidad usada para la configuración. Al hacer esta conversión nos da una tasa de 100 bit/ $\mu$ s, lo cual es lo mismo que 12.5 Byte/ $\mu$ s. Ahora, las tramas son de 60B + 8B preámbulo + 4B FCS = 72 Bytes, por lo tanto se demora 5,76  $\mu$ s en transmitir una de ellas.

Si bien la demora de enviar una trama sobre una interfaz de 100Mbps son 6  $\mu$ s aproximadamente, el sistema que es la fuente de información, tiene una latencia dependiendo de la carga de procesamiento que tenga, lo cual puede redundar en *jitter*. Si no tenemos en cuenta este posible *jitter*, la transmisión o recepción de la trama podría (pensando en un peor caso hipotético) demorarse en el orden de la latencia del sistema.

Nota: Vamos a despreciar los retardos que pueden presentarse en el medio físico por estar en distancias de cableados de una LAN, así como cualquier retardo que se puede generar en la placa de red.

El Xenomai tiene un conjunto de herramientas para su testeo y en particular una para medir la latencia del sistema, llamada *latency* [28]. Luego de varios ensayos, cargando al sistema, el peor registro que obtuvimos fue el siguiente:

```
RTH|----lat min|----lat avg|----lat max|---overrun|---msw|---lat best|--lat worst
RTD|  -2.146|      -1.720|    3.432|    0|    0|   -2.375|    21.339
RTD|  -2.129|      -1.621|    3.044|    0|    0|   -2.375|    21.339
RTD|  -2.268|      -1.729|    4.863|    0|    0|   -2.375|    21.339
RTD|  -2.131|      -1.714|    2.210|    0|    0|   -2.375|    21.339
RTD|  -2.213|      -1.679|    4.911|    0|    0|   -2.375|    21.339
RTD|  -2.290|      -1.731|    2.983|    0|    0|   -2.375|    21.339
RTD|  -2.217|      -1.734|    4.665|    0|    0|   -2.375|    21.339
RTD|  -2.132|      -1.701|    3.525|    0|    0|   -2.375|    21.339
RTD|  -2.135|      -1.731|    3.034|    0|    0|   -2.375|    21.339
RTD|  -2.185|      -1.718|    4.170|    0|    0|   -2.375|    21.339
RTD|  -2.277|      -1.690|    2.998|    0|    0|   -2.375|    21.339
RTD|  -2.184|      -1.665|    3.480|    0|    0|   -2.375|    21.339
```

RTD	-2.169	-1.717	3.963	0	0	-2.375	21.339
RTD	-2.342	-1.702	2.708	0	0	-2.375	21.339
RTD	-2.296	-1.690	4.119	0	0	-2.375	21.339
RTD	-2.247	-1.680	2.223	0	0	-2.375	21.339
RTD	-2.256	-1.701	4.171	0	0	-2.375	21.339
RTD	-2.329	-1.666	5.494	0	0	-2.375	21.339
RTD	-2.263	-1.522	6.455	0	0	-2.375	21.339
RTD	-2.321	-0.881	10.199	0	0	-2.375	21.339
RTD	-2.253	-1.434	7.433	0	0	-2.375	21.339
RTT	00:02:07 (periodic user-mode task, 100 us period, priority 99)						

Donde para una tarea (o hebra) lanzada con un período de reiteración de  $100\mu\text{s}$ , en espacio de usuario (no de *kernel*) pero con la prioridad más alta (*priority* 99 [29])- que simula de buena forma las tareas *real-time* que serán lanzadas por el HAL del LinuxCNC- la latencia es de  $22\mu\text{s}$ .

Si bien el RTnet tiene en el mecanismo de calibración una forma de compensar todas posibles fuentes de latencia, siempre estará presente el *jitter*, y la duración del *time slot* debe tenerlo en cuenta [30]. Para este cálculo diremos que la latencia se traduce totalmente en *jitter*, para estar seguros que cubrimos sus efectos. Entonces este sería el peor caso y para cuantificar el efecto del *jitter* en este caso, supongamos que la transmisión hecha por dos nodos, Nodo 1 con un *jitter* del mismo orden que el Nodo 2. Sucedería lo que se muestra a continuación:

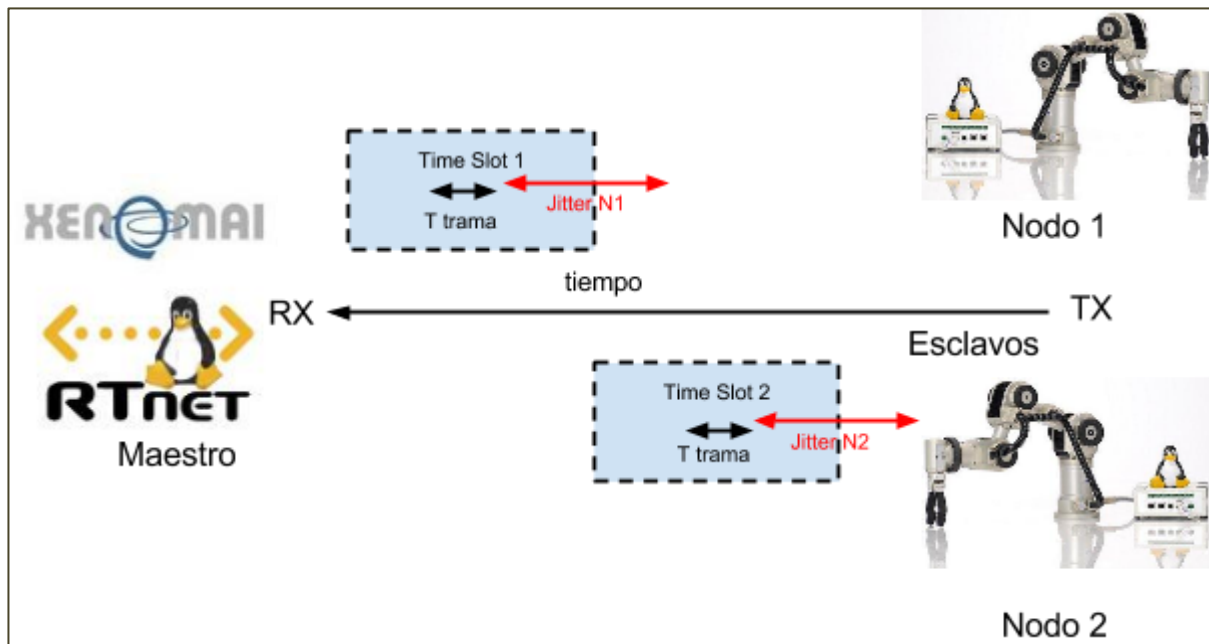


Figura 3.5: Envío de tramas en dos *time slots* al Maestro.

Supongamos que en la etapa de transmisión de la trama, el Nodo 1, podría llegar a tener un *jitter* de  $22\mu\text{s}$  en el comienzo del *time slot* 1, lo mismo para el Nodo 2, pero en el comienzo *time slot* 2. En la figura se muestra en rojo el efecto del *jitter*, ya que tanto el comienzo de la transmisión de la trama como la referencia de comienzo del *time slot* son afectados por *jitter* en la latencia (ver proceso de calibración en RTnet [30]). Entonces la duración del *time slot* debe cubrir el doble de la latencia, que son  $44\mu\text{s}$ . A esto le tenemos que sumar el tiempo de duración de la trama que dijimos es de  $6\mu\text{s}$ .

Así para que el *time slot* soporte el peor caso, para transmitir tramas de 72 Bytes, un *time slot* de  $50\mu\text{s}$  nos asegura un envío y recepción correcta de nuestra información, por tanto  $\text{TDMA\_OFFSET}=50$ .

### 3.2.2.1.2 Tiempo de ciclo de la trama TDMA:

El otro parámetro necesario para poder configurar el TDMA de RTnet, es la duración de la trama TDMA. Sabiendo que los *time slots* van a ser de  $50\mu\text{s}$ , va a depender la cantidad de nodos que tengamos en nuestra red o bus y las limitaciones de tiempo que exija nuestra aplicación.

En caso de una máquina herramienta CNC de 5 ejes, más 1 o 2 nodos para sensores o interruptor de emergencia, con un TDMA\_CYCLE=350μs es suficiente para tener una trama de 7 *time slots* de 50μs cada uno.

Pero, luego analizaremos que implicancias se observaron en los ensayos con distintos valores de TDMA\_CYCLEs en nuestra arquitectura, cuando comenzamos a mandar paquetes CANopen.

### **3.2.3 Etapa de integración RTnet y CanFestival**

Como se mencionó antes, la estructura de las librerías CanFestival, tienen previsto la implementación de drivers con una estructura de primitivas y funciones, que para ser integrado, hay que codificar orientado a la interfaz y medio físico al cual van a ser enviadas las tramas CANopen.

La contraparte mayor que tiene el proyecto CanFestival es su falta de documentación, en particular para desarrollar aplicaciones. De todas formas, al tener disponible el código y algunos ejemplos, se puede lograr comprender lo suficiente para desarrollar una implementación.

Dentro de la documentación que existe [31], tenemos un esquema que sirve para conceptualizar todos los elementos de esta librería y su interacción. El primer esquema muestra cómo interacciona la librería con el resto del código del sistema:

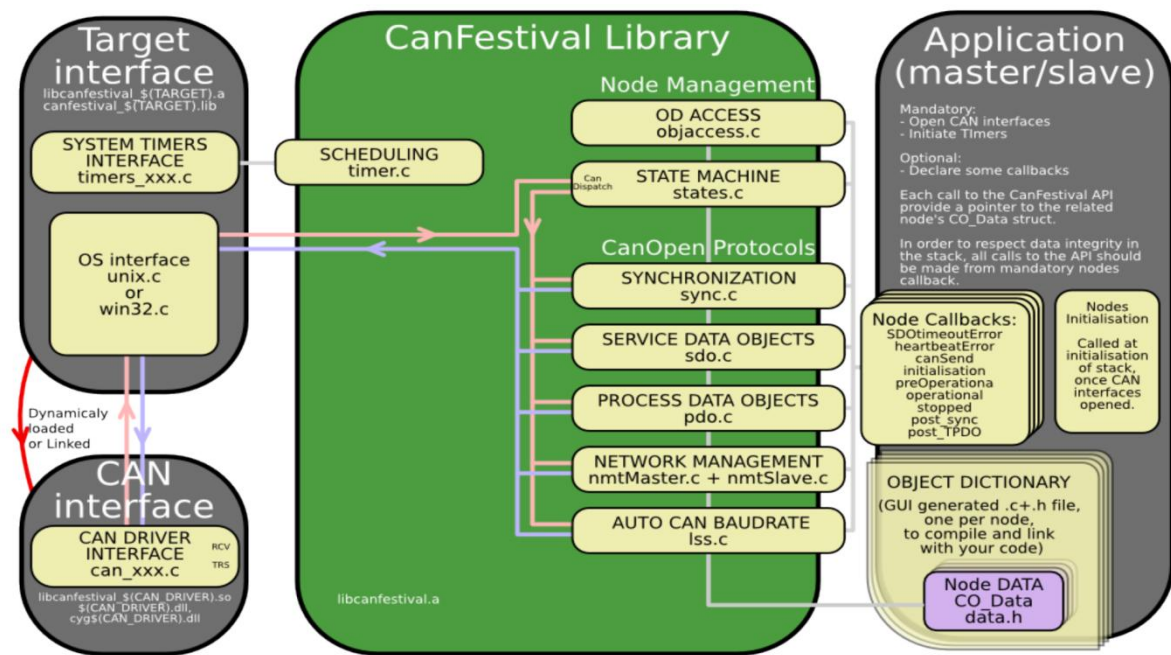


Figura 3.6

En la figura 3.6 (ver [31]) se observa lo que sería la implementación de un maestro o esclavo CANopen, las llamadas que hace a la librería CanFestival, el flujo de llamadas a las funciones que comunican con el *kernel* del sistema operativo (en nuestro caso Xenomai que es tipo unix.c con timers\_xeno.c según CanFestival) y este enlaza dinámicamente con la interfaz CAN, en nuestro caso el driver que desarrollamos es el can\_rtinet.c.

De esta forma, previo al desarrollo de una aplicación, es necesario codificar un driver para la interfaz RTnet. Nuestra implementación se compone de 6 funciones como se muestra en el esquema a continuación:

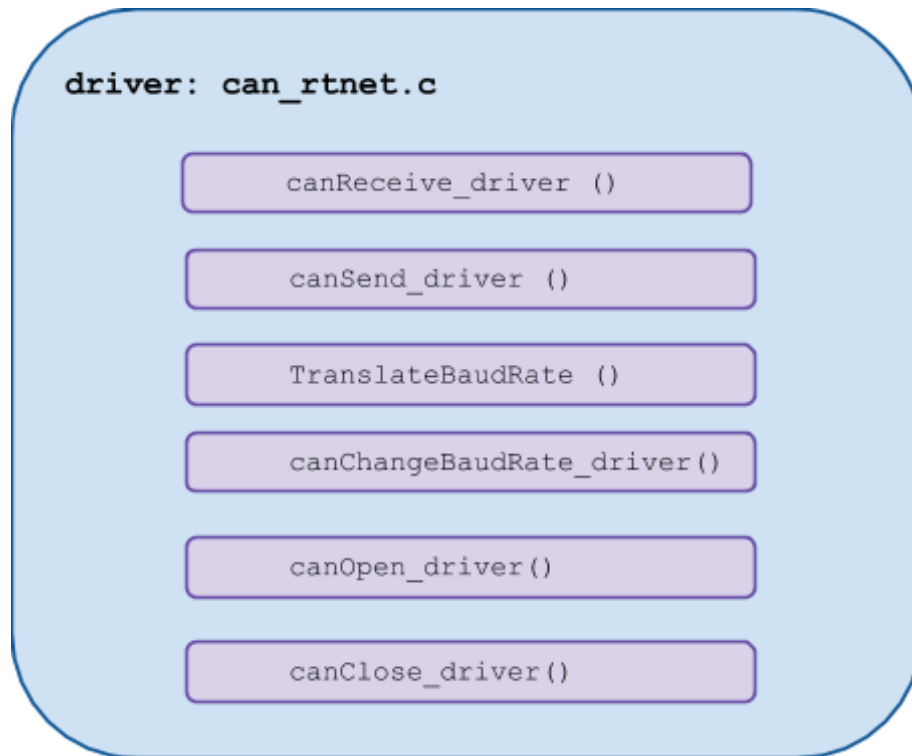


Figura 3.7: Estructura del driver can\_rtinet

El driver tiene previsto usando macros, verificar si el *kernel* del sistema es RTAI/Xenomai o un *kernel* standard. Tendrá así asignadas las primitivas correspondientes para el manejo de *sockets* según el *kernel*.

Luego se implementan las funciones:

- `canRecive_driver`: recibe tramas de la interfaz RTnet, que luego son enviadas a la librería CanFestival, donde se analiza que tipo de comunicación corresponde (*Service Data Objects* (SDO), *Process Data Objects* (PDO), *Network Management* (NMT), etc) para que se tomen la acciones pertinentes a nivel de aplicación y sobre el diccionario de objetos.
- `canSend_driver`: envía las tramas una vez procesada por las librerías CanFestival a la interfaz RTnet en formato RTCANopen.
- `TranslateBaudRate`: esta función está pensada para una interfaz CAN serial, nuestra implementación acepta los parámetros de entradas pero no realiza ninguna operación.
- `canChangeBaudRate_driver`: esta función está pensada para una interfaz CAN serial, nuestra implementación acepta los parámetros de entradas pero no realiza ninguna operación.

- `canOpen_driver`: esta función abre un *socket* RAW y devuelve el *file descriptor* del *socket* para que sea usado por la funciones `canRecive_driver` y `canSend_driver`.
- `canClose_driver`: cierra el *socket* RAW.

Nótese que las funciones de apertura y cierre del *socket*, como la de envío y la de recepción de tramas, terminan con “\_driver”. La librería CanFestival se encarga de mapear las llamadas a nivel de aplicación a `canRecive`, `canSend`, `canOpen` y `canClose` a las funciones correspondientes al driver que le fue pasado en el *script* “configure” previo a la compilación de la librería.

Para que nuestro driver fuera compilado con el resto de la librería, hubo que ponerlo dentro del *path* `../Canfestival/drivers/can_rtnet/` y generar un Makefile que se encargue de su compilación e instalación. Los detalles de la instalación se encuentran en el Anexo.3.

### 3.2.4 Etapa de integración CanFestival y LinuxCNC

Luego de las etapas anteriores, se llegó a donde se integra todo el trabajo previo en un componente para el HAL del LinuxCNC, que nos permitirá conectar componentes HAL con un nodo remoto vía CANopen transportado sobre RTnet.

Como paso previo se genera el diccionario de objetos del maestro mediante el editor que integra el paquete CanFestival, en su versión para GNU/Linux es una GUI basada en WxPython llamada `objdictedit.py` [31].

Haremos una breve introducción al HAL cuya documentación viene integrada dentro de la aplicación LinuxCNC, y también se puede acceder desde la página del proyecto [32]. La estructura del código de un componente tiene la siguiente forma:



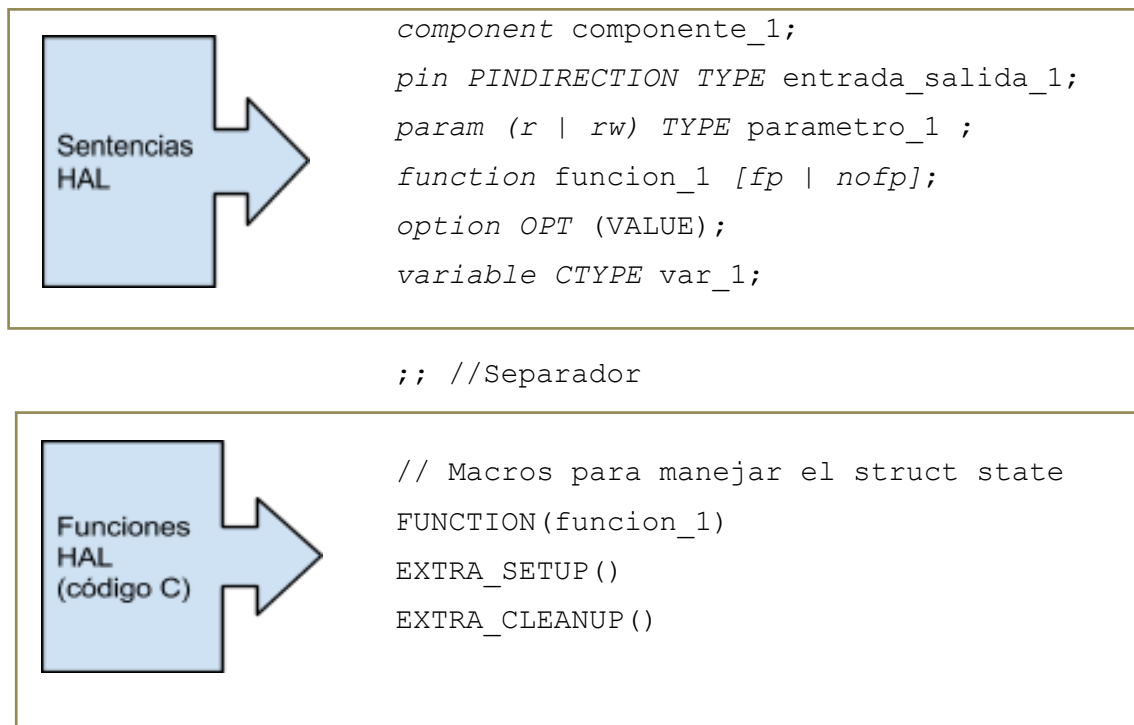


Figura 3.7

Dentro de las sentencias HAL:

1. Primero se da nombre al componente.
2. Se definen los pines de entrada/salida/entrada-salida con su tipo (puede ser un bit, byte, integer, etc).
3. Se definen parámetros que pueden ser seteados como *read* ó *read/write*.
4. A continuación se definen las funciones que hacen al componente. Estas procesan las entradas y pasan su valor a las salidas.
5. Los componentes tienen varias opciones, como por ejemplo si van a ser componentes de espacio de usuario o un módulo de *kernel*. En particular le decimos al HAL si nuestro componente van a usar las funciones de EXTRA\_SETUP y EXTRA\_CLEANUP. En nuestra implementación esto juega un rol importante.
6. También se pueden declarar variables para ser usadas en las funciones, excepto en la EXTRA\_SETUP. Para las variables de la función EXTRA\_SETUP se usa la porción de código C, y para pasarle valores en el momento de instanciar el componente se debe usar la función RTAPI\_MP\_ARRAY\_INT.

Otra precisión importante tiene que ver con la diferencia que hay entre las funciones y el EXTRA\_SETUP y EXTRA\_CLEANUP. Las funciones son lanzadas en un *thread* que se repite en un período configurable de nanosegundos. La función EXTRA\_SETUP se ejecuta solo una vez por instancia del componente.

Entonces para implementar el “*setup*” del CanFestival dentro del componente HAL, se optó por realizarlo dentro de la función EXTRA\_SETUP. De esta forma el envío y recepción de tramas CANopen se realiza dentro de *threads* generados por la librería CanFestival, que es como está pensado se haga. Un flujo-grama de los *threads* que se lanzan y que tareas se realizan en cada uno se puede observar en la siguiente figura:

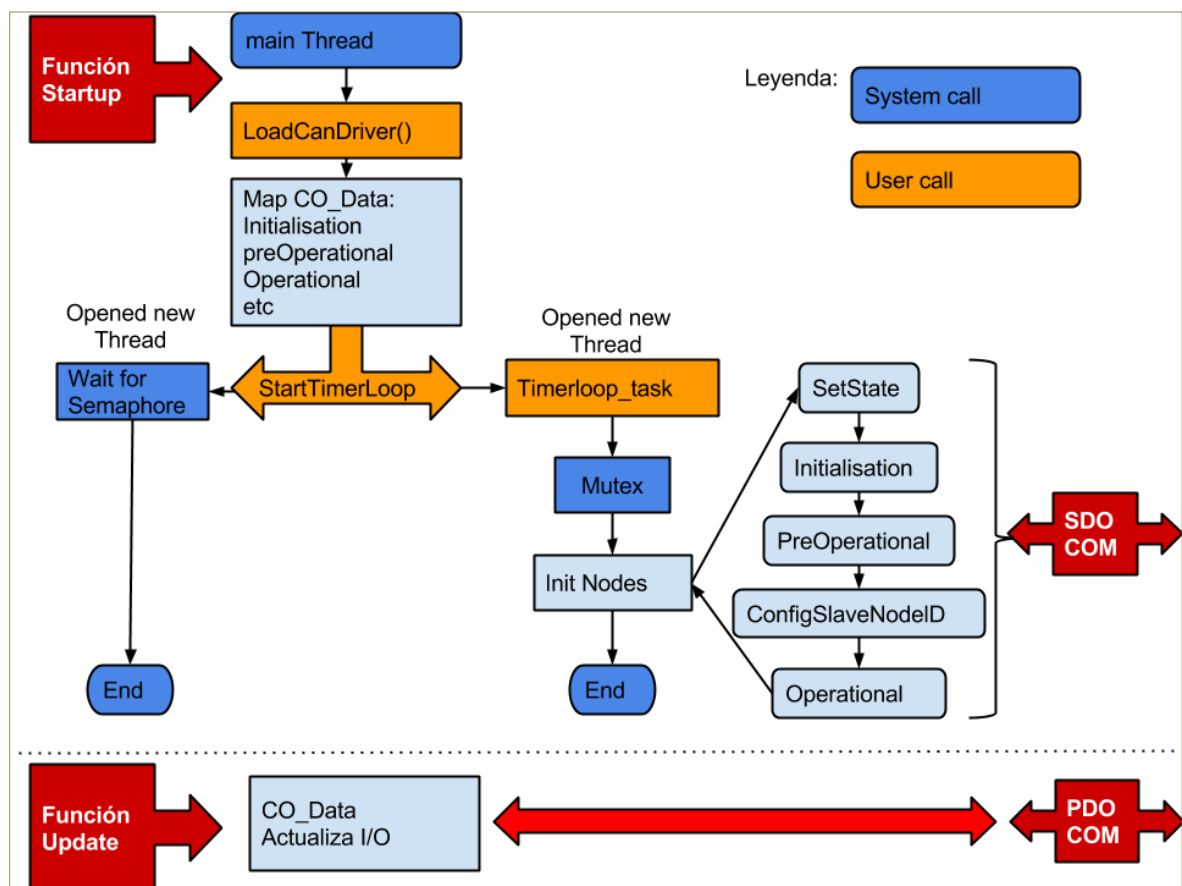


Figura 3.8

Según la figura 3.8, el *thread* principal comienza por hacer el enlace dinámico al driver can\_rtnet (que queda como una librería libcan\_rtnet.so) ejecutando LoadCanDriver(). Seguidamente se mapean a la estructura que define el diccionario de objetos CANopen del nodo, las funciones que se van a encargar de los distintos estados de

funcionamiento. De esta forma, se puede lanzar el `StartTimerLoop()`, donde se lanzan dos *threads*, uno que lanza toda la inicialización del nodo maestro y configura el nodo esclavo, hasta dejar ambos en estado “operacional”. El otro *thread* se queda esperando la finalización del primero y ejecuta cualquier tarea con la que se desea finalizar la función `EXTRA_SETUP`.

Nótese que al dejar a ambos nodos en estados operacional, ya se comunicaron vía SDO todas la configuraciones de los COBID necesarios, por tanto los nodos quedan transmitiendo los PDOs y SYNC como se hayan configurado. En nuestra implementación, los PDOs se transmiten con los SYNCs que genera el maestro.

Luego que el componente “canopen” ha realizado su *setup*, se lanza la función *update* en un nuevo *thread* desde el *halcmd* del LinuxCNC, con un período de 1 milisegundo. Donde lo único que hay que hacer es escribir los datos que se encuentran en las entradas en el COBID del diccionario de objetos que tenga mapeado un *Transmit Process Data Object* (TPDO) y leer del diccionario de objetos el COBID que tenga mapeado un *Recive Process Data Object* (RPDO) para escribir la salida correspondiente.

Así, se puede conectar las entradas y salidas del componente “canopen” a las de un componente que controle el esclavo con el que este comunica.

Si se desea manejar varios esclavos, se deben configurar una instancia del componente por cada uno.

### **3.3 Escalabilidad de la solución Maestro/Controlador**

La escalabilidad de la solución va a estar limitada a la cantidad de nodos esclavos que se deseen controlar y la limitación de temporización que exija la aplicación. Ya que estos factores limitan el tiempo de ciclo y *offset* del RTnet. Estos serían los factores más importantes.

Ahora teniendo un hardware que tenga un *jitter* en la latencia especificado, como se mostró, nos permitiría usar *time slots* más cortos en duración y tener un tiempo de ciclo de duración suficiente para manejar el control de un número importante de esclavos.

Un número mayor de pruebas serían necesarias para poder realizar un análisis estadístico de *jitter* del sistema y poder afinar los valores de duración de *time slot* y tiempo de ciclo, que repercuten en la cantidad de nodos a ser soportados por el bus.

### 3.4 Posibles Mejoras

- Durante los ensayos realizados se encontró que la librería CanFestival no tiene un tratamiento de error al recibir tramas con los SDO *abort codes*. Creemos importante mejorar la librería sdo.c del CanFestival que es donde se tratan las tramas SDO, para incluir estos tratamientos, así depurar rápidamente cualquier error en la configuración del CANopen realizada en una aplicación. Así ayudar a acortar el tiempo de puesta en marcha de nuevas aplicaciones.
- Nuestro driver para CanFestival, can\_rtnet.c, envía sus tramas a la dirección MAC de broadcast, dejando que el Node ID resuelva el direccionamiento. Vimos que las librerías de CANopen usadas, dejan que la capa de enlace CAN resuelva este direccionamiento. Por lo tanto para una implementación de más de un esclavo, se debe mapear el Node ID a la dirección MAC del nodo destino de la trama e introducirlo en la función canSend\_driver. Alcanzaría con un arreglo estático “macs” cuyo índice sea el Node ID. El archivo tdma.conf de RTnet (que es la capa encargada del direccionamiento MAC) contiene las MACs de los nodos, entonces leyéndolo se podría cargar este arreglo donde el Node ID puede ser un comentario del mismo archivo.
- Realizar la implementación en una SBC (Single Board Computer). Lo cual hace aún más sencillo la instalación en ambientes menos limpios, donde el chasis de un PC puede ocupar espacio o no tener las condiciones ambientales para su correcto funcionamiento, mientras que un SBC se puede ubicar dentro de una caja o gabinete pequeño donde sí está controlado el ambiente.

## CAPITULO 4. Esclavo

### 4.1 Arquitectura XMOS:

Para la implementación de los dispositivos remotos optamos por una novedosa arquitectura de procesadores multitarea ofrecida por la firma XMOS. Esta arquitectura se presenta como una solución novedosa y atractiva con atributos de lógica programable, herramientas para la gestión de procesos en paralelo, alta potencia de procesamiento, bajo costo y extensas librerías de código abierto.

La arquitectura XMOS pone especial énfasis en un alto poder de procesamiento multitarea; de acuerdo con las especificaciones un procesador de rango medio como el XS1-L16A-128 es capaz de alcanzar 1000 MIPS y distribuirlos equitativamente en hasta 16 núcleos lógicos ejecutando tareas concurrentemente.

Además de esto la arquitectura incorpora estructuras de hardware de interfaz que le permiten interactuar con señales externas de forma rápida y estable (libre de *jitter*); por ejemplo *time-stamping* en hardware, serialización en hardware, la captura y generación de flancos pueden ser encargados al hardware sin recurrir a *polling* obteniendo un desempeño altamente determinístico además de veloz.

La arquitectura incluye un sistema eficiente para la gestión de los ciclos de instrucción: cuando un núcleo entra en espera por un evento la ejecución es entregada a otros núcleos sin entrar nunca en bucles de espera ineficientes. Cuando el evento esperado ocurre, la ejecución es inmediatamente devuelta -con latencia máxima de 100 ns- al núcleo que estaba esperándolo.

Fundamental para el aprovechamiento del procesamiento en paralelo es una estructura de comunicaciones rápida entre núcleos mediante canales unidireccionales que resuelve los problemas de sincronización de datos entre los núcleos lógicos. Estos canales de comunicación están implementados en hardware y además de comunicar núcleos de una misma pastilla pueden establecerse canales entre distintas pastillas de un mismo procesador y también entre pastillas de distintos procesadores. XMOS ofrece procesadores integrados por una o más pastillas con características similares, una forma modular de escalar la potencia del procesador.

XMOS se ofrece como una arquitectura para soluciones embebidas pero sigue una filosofía diferente a la actualmente tradicional de incluir múltiples componentes de hardware embebidos en cada chip como ser UARTs, interfaces SPI, temporizadores, generadores PWM, etc. Por el contrario la filosofía es **programar** los componentes de hardware que se requieran escribiendo código para ser ejecutado en núcleos independientes. Esta solución para la implementación de hardware embebido resulta sumamente flexible; XMOS ya ofrece amplias bibliotecas de código abierto para la implementación de componentes estándar como UART, SPI, PWM, *timers*, etc. Cada núcleo dispone de estructuras de entrada/salida que permiten resolver todas estas funciones con el mismo desempeño de un módulo de hardware dedicado pero teniendo una gran ventaja en cuanto a flexibilidad de los componentes (alterando su código) y también customización del procesador (eligiendo los componentes deseados).

## 4.2 Implementación del Bus de Campo CANopen/RTnet

El esclavo se implementó sobre un kit de desarrollo estándar “slice-Kit” de la firma XMOS que incluye como interfaz de red una placa “Ethernet slice”. La placa madre del kit está basada en el procesador XS1-L16-128-QF124 de 16 núcleos y 1000 MIPS y la placa Ethernet está basada en el chip LAN8710A-EZK con capacidad full-duplex a 100 Mbps.

Para la comunicación con el Maestro el dispositivo utiliza el protocolo de arbitraje de red RTnet [2] sobre una conexión estándar Ethernet a 100Mbps y el protocolo de comunicaciones CANopen de acuerdo al estándar CiA 301 [33]. La funcionalidad implementada en el dispositivo está en concordancia con el perfil estándar de CiA DS402 [34] para controladores de movimiento.

Se incorporaron adicionales por sobre el perfil CiA DS402 para el reporte de posición de un segundo eje. El dispositivo esclavo resultante ofrece acceso en tiempo real a través del Bus de Campo las siguientes funciones:

Control de velocidad de giro de un servomotor (voltaje aplicado).

- Reporte de posición y velocidad de su eje.
- Reporte de posición de un segundo eje independiente.

- Control de 4 salidas digitales.
- Reporte de estado de 1 entrada digital.

#### 4.2.1 Gestión de acceso y formato de trama RTnet:

La gestión de acceso al medio que propone el protocolo RTnet está basada en la subdivisión en *Slots* (ranuras) de períodos de tiempo de duración fija establecida por el maestro RTnet. El maestro introduce en el Bus señales en forma de tramas Sync para marcar el inicio y el final de los períodos de tiempo. Cada trama Sync señala simultáneamente el final de un período y el comienzo de otro, repitiendo la secuencia indefinidamente en el tiempo:

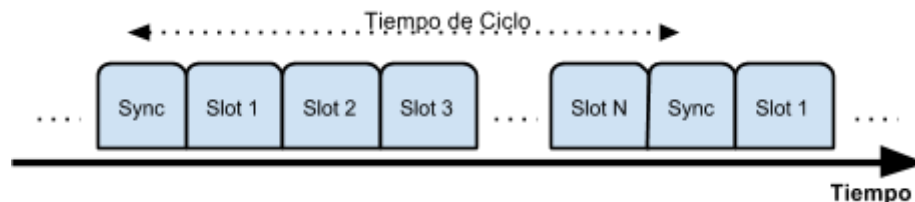


Figura.4.1

El tiempo transcurrido entre sucesivas tramas Sync se llama tiempo de ciclo y en este esquema el ciclo abarca una cantidad N de Slots o subdivisiones. El protocolo sigue unas reglas básicas para gestionar el uso de los Slots:

La propiedad de los Slots se establece de común acuerdo entre maestro y esclavos.

- Solamente el propietario de un Slot tiene permitido transmitir dentro de su duración.
- El protocolo no permite la ocurrencia de colisiones de tramas.
- Tanto maestro como esclavos pueden tener asignados uno o más Slots para introducir tramas al Bus.
- Las tramas Sync transportan únicamente información relativa a la sincronización del Bus, no se transmiten datos de otro tipo en una trama Sync. Al igual que los esclavos, el maestro también necesita de uno o más Slots para comunicar datos a través del Bus.

Todas las tramas introducidas en el Bus tanto por el maestro como por los esclavos se ajustan al formato de trama estándar Ethernet. La organización de los campos no difiere

en absoluto de una trama Ethernet común y corriente [2] pudiéndose utilizar switches entre maestro y esclavos RTnet:

Preámbulo	SFD	MAC Destino	MAC Origen	Ethertype	Payload	CRC
8 Bytes	1 Byte	6 Bytes	6 Bytes	90-23	46...1500 Bytes	4 Bytes

Figura 4.2

Nota: los campos de preámbulo (7 octetos), *start-of-frame-delimiter* (1 octeto) y CRC (4 octetos) son normalmente introducidos automáticamente en la trama por el chip PHY del interfaz.

El código de esta implementación filtra las tramas recibidas en función del valor del campo Ethertype, sólo los tipos de trama 08-00 (ICMP), 08-06 (ARP), 90-21 (RTmac), 90-22 (RTcfg) y 90-23 (RTCANopen) son interpretados. Todas las tramas recibidas con campo Ethertype de 90-23 son interpretadas como portadoras de una trama CANopen en su campo *Payload* y son desencapsuladas.

## 4.2.2 Implementación del protocolo de red

La implementación de RTnet en la arquitectura XMOS se construyó desde cero para este proyecto, escribiendo varias funciones y un servicio en lenguaje XC -versión de C adaptada a esta arquitectura-. Destaca el servicio *rtnet\_sync* encargado de mantener sincronía con las tramas Sync periódicamente transmitidas por el maestro RTnet. Este servicio se ejecuta concurrentemente con el resto de las tareas desempeñadas por el dispositivo ocupando 1 núcleo lógico del procesador.

Con *rtnet\_sync* se adoptaron dos conceptos básicos de la teoría de sistemas operativos de tiempo real (RTOS): el agendado de acciones y su priorización. La acción fundamental del servicio que es regular el acceso al medio por parte del esclavo debe respetar determinísticamente la temporización impuesta por el maestro a la vez de mantener prioridades en el uso de la porción del medio asignada.



Constantemente *rtnet\_sync* espera por tramas Sync provenientes desde el maestro RTnet y por solicitudes de transmisión de tramas CANopen desde el servicio correspondiente -ejecutando en otro núcleo lógico-. Las tareas atendidas por este servicio se pueden resumir en las siguientes:

Cronometraje de tramas Sync recibidas desde el maestro RTnet con resolución de 10ns.

- Calibración de tiempo de transporte de tramas entre maestro RTnet y este dispositivo promediando 100 muestras con resolución de 10nS.
- Desencapsular y pasar al servicio CANopen tramas RTnet con campo Ethertype específico (9023).
- Recibir, empaquetar y transmitir tramas desde el servicio CANopen.
- Atender solicitudes ARP para resolución de direcciones.
- Atender solicitudes de PING.
- Agendar **todas** las transmisiones para que se efectúen en correcta sincronía con los Slots de tiempo asignados a este dispositivo y respetando el orden de prioridades.

Entre las funciones accesorias escritas especialmente para esta implementación cabe nombrar a 3 no por complejidad sino por funcionalidad:

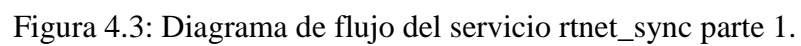
- *armar\_trama\_cal\_req()*. Encargada de armar las tramas de solicitud de respuesta temporizada desde el maestro RTnet, utilizada para calibrar el tiempo de transporte de las tramas entre el maestro y este dispositivo. Cada dispositivo en la red realiza su propia calibración.
- *eleva\_trama\_canopen()*. Encargada de desencapsular la trama RTnet, rearmarla en el formato apropiado y enviarla al servicio CANopen el cual corre en otra hebra.
- *bajar\_trama\_canopen()*. Encargada de encapsular las tramas recibidas del servicio CANopen en tramas RTnet para ser agendadas en la cola de transmisión del servicio *rtnet\_sync*.

Para el acceso al chip PHY esta implementación hace uso del módulo *ethernet\_server* [35] provisto por XMOS en modalidad “full”. En esta modalidad el

módulo ocupa 5 núcleos lógicos para ofrecer servicios de transmisión y recepción *full-duplex* y filtrado de paquetes Ethernet a 100Mbps en un máximo de 2 interfaces -ésta implementación utiliza 1 sólo interfaz-.

#### **4.2.2.1 Fundamentos del código**

El corazón de la implementación del protocolo de red es el servicio *rtnet\_sync* que asegura la disciplina del dispositivo en su acceso al medio Ethernet. La estructura fundamental para lograr la temporización de tramas es un buffer de tramas denominado *slot\_agenda*. Este buffer está organizado como un arreglo con tantas entradas como Slots tiene asignados el dispositivo. Dado que la transmisión de tramas debe respetar una temporización precisa, la rutina, servicio o aplicación que deba transmitir una trama deberá “agendar”. Para “agendar” la transmisión de una trama en el Slot N se debe llenar la entrada N del arreglo *slot\_agenda* y el servicio *rtnet\_sync* se encargará de transmitirla y vaciar la entrada.



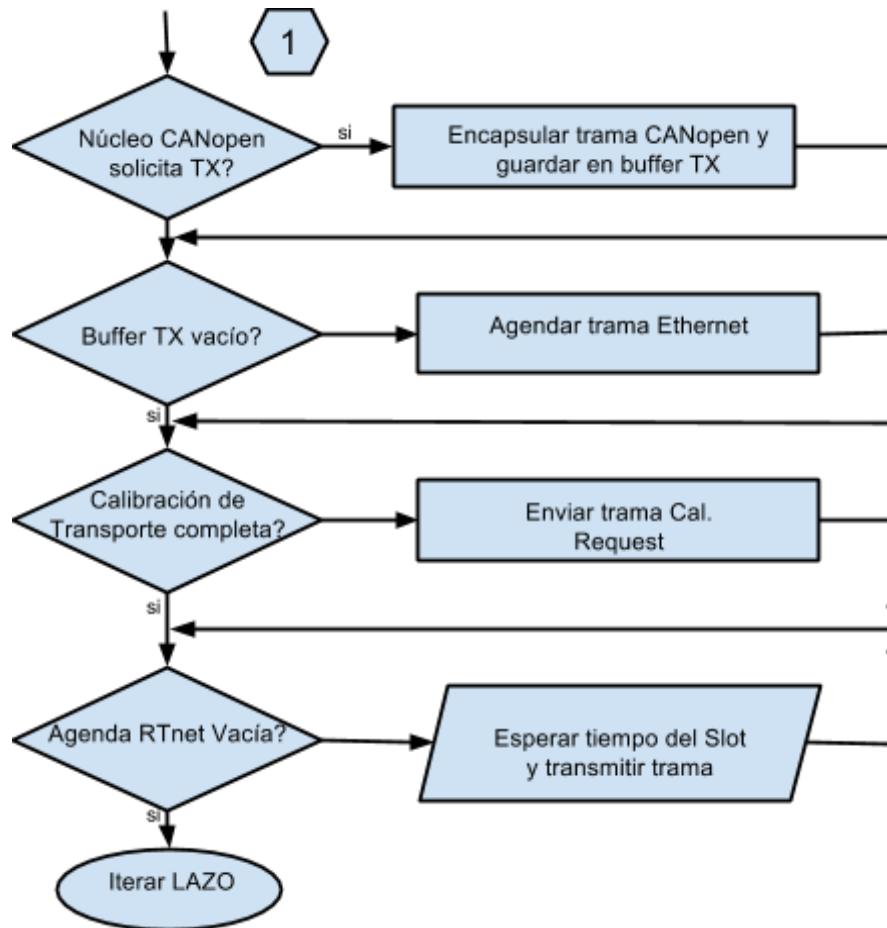


Figura 4.4: Diagrama de flujo del servicio *rtnet\_sync* parte 2.

#### 4.2.2.2 Organización de Núcleos

Un total de 6 núcleos lógicos del procesador son destinados a la implementación del protocolo de red RTnet. Cada uno de los núcleos lógicos ejecuta cíclicamente una rutina que atiende una tarea específica. Los núcleos utilizan canales implementados en el hardware del procesador para transferir datos mutuamente.

El núcleo ocupado por el servicio *rtnet\_sync* utiliza un total de tres canales de comunicación: uno para la transmisión de tramas hacia el servicio de transmisión Ethernet, otro para la recepción de tramas desde el servicio de recepción Ethernet y un tercero bidireccional para “elevar” o “bajar” tramas CANopen hacia y desde el servicio *canopen*.

Las funciones para acceso *full-duplex* al medio Ethernet ocupan los restantes 5 núcleos organizados de la siguiente forma:

Servicio de transmisión Ethernet: Armado de tramas con preámbulo y CRC para ser transmitidas

1. Servicio de recepción Ethernet: desarmado de tramas Ethernet recibidas
2. Transferencia de tramas al chip PHY a través del interfaz MII
3. Recepción de tramas desde el chip PHY a través del interfaz MII
4. Filtrado de tramas en función con campo Ethertype

Un esquema simplificado de la comunicación entre los núcleos integrantes de la implementación de RTnet se presenta a continuación:

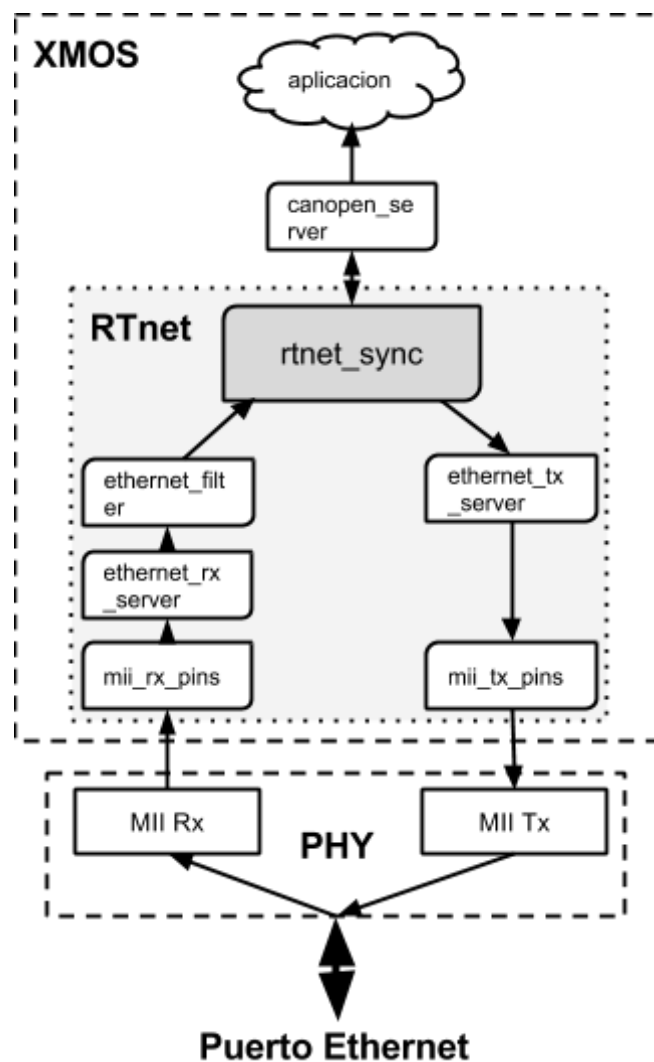


Figura 4.5

### **4.2.3 Implementación del protocolo de comunicaciones CANopen**

XMOS ofrece para su plataforma de procesadores una librería (módulo) de código abierto apta para implementar los protocolos y perfiles de comunicaciones CANopen de acuerdo con la especificación CiA 301 [33]. Posee la funcionalidad para interpretar y generar todos los tipos de tramas CANopen, establecer modos de operación, sincronizar operaciones con tramas SYNC, mapear un diccionario de objetos y generar tramas “*heartbeat*” y de emergencia de acuerdo con el estado funcional del dispositivo esclavo.

Para la implementación de nuestro bus de campo hemos acoplado la librería estándar CANopen ofrecida por XMOS a nuestro servicio *rtnet\_sync* para de esta forma transportar tramas CANopen en tramas Ethernet con arbitraje RTnet.

#### **4.2.3.1 Modificaciones efectuadas a la librería CANopen**

Con el fin de optimizar el acceso al diccionario de objetos CANopen por parte de la rutina de aplicación del dispositivo se agregaron funciones de acceso directo a través de punteros. Estas funciones resuelven la lectura y escritura al diccionario de objetos tipo Byte, Short e Integer de 8, 16 y 32 bits respectivamente.

Toda implementación de un dispositivo CANopen requiere la confección de su propio diccionario de objetos. La librería CANopen ofrecida por XMOS incluye una herramienta para obtener el diccionario de objetos en forma de un archivo “.h” a incluir en el código. Esta herramienta requiere un archivo en formato EDS (Electronic Data-Sheet) y para obtener este archivo EDS se utilizó la aplicación Vector-CANeds (o CANeds). Con la aplicación CANeds se incorporan al diccionario de objetos los elementos necesarios para la implementación de las funciones requeridas.

#### 4.2.3.2 Controlador de movimientos CANopen-DS402

El controlador de movimiento CANopen ocupa el lugar de capa de aplicación en la implementación del dispositivo esclavo. El servicio **aplicación** fue escrito a tal efecto, este ocupa 1 núcleo del procesador, emplea 3 canales de comunicación inter-núcleo y lleva a cabo las funciones de mayor jerarquía en la implementación del dispositivo controlador de movimientos, a saber:

- Control de los estados Detenido, Listo, Encendido, Fallo del controlador.
- Control de los modos de operación del controlador.
- Recibir desde Maestro CANopen comandos de gestión de estado del controlador.
- Actualizar salidas PWM y salidas digitales de acuerdo a cambios en los objetos correspondientes del diccionario de objetos.
- Actualizar diccionario de objetos de acuerdo a cambios en entradas de encoders y entradas digitales comunes.

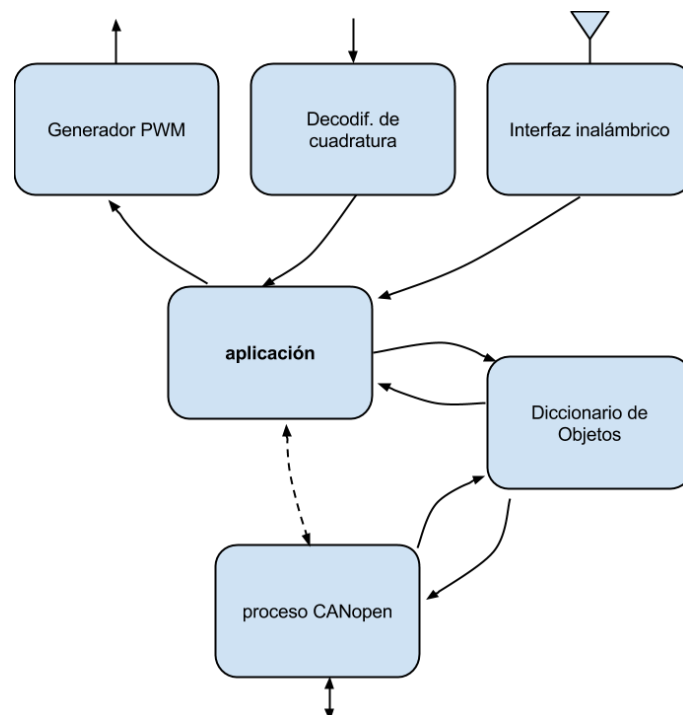


Figura 4.6: Esquema funcional de la implementación del controlador DS-402.

La gestión de modos de operación y estados del controlador es implementada como una máquina de estados atendiendo eventos provenientes desde el maestro CANopen y

desde entradas de señal -señal de fallo p.ej.-. El protocolo de comunicaciones CANopen cubre ampliamente los requerimientos de señalización remota y chequeo de condiciones necesarios para el comando de un controlador de movimientos. El servicio *aplicacion* hace uso de un canal de comunicaciones bidireccional con el servicio *canopen* para recibir instrucciones desde y reportar hacia el maestro CANopen remoto.

Para el control de las salidas PWM el servicio hace uso de un canal de comunicación inter-núcleo hacia un servicio específico de generación de señales PWM: *pwmsinglebitport*. Simultáneamente se realiza la actualización de la señal de dirección correspondiente con cada salida de comando PWM. La actualización del valor de salida PWM y su señal de dirección son efectuados con la detección de cambios en los objetos asociados a cada salida PWM.

La lectura de posición del eje controlado (eje acoplado al movimiento del motor) se lleva a cabo a través del servicio específico *do\_multi4\_qei* que es una versión modificada para este proyecto del servicio estándar ofrecido por XMOS para la lectura de señales en cuadratura, generalmente provistas por los codificadores rotativos. Este servicio lleva cuenta de la ubicación de hasta 4 codificadores, tiene soporte para señal índice y reporta las posiciones a través de un canal de comunicación inter-núcleo.

Para la lectura de posición del segundo eje se implementó un enlace inalámbrico de 1 Mbps en banda de 2.4 GHz basado en módulos estándar nRF24L01. Este enlace posibilita la localización del segundo eje sobre una estructura móvil sin limitaciones de cableado. Para la utilización del módulo inalámbrico se escribieron funciones accesorias de configuración, lectura y escritura desde y hacia el módulo inalámbrico que hacen uso de la librería *spi\_master* provista por XMOS.

#### **4.2.3.3 Organización de Núcleos**

Un total de 4 núcleos lógicos son utilizados en la implementación del controlador de movimientos CANopen, entre éstos cuenta el servicio *aplicacion* cuya funcionalidad se ha descrito en los párrafos anteriores. Una breve descripción de la separación de tareas entre los núcleos se muestra a continuación:



Servicio *aplicacion* controla máquina de estados del dispositivo, atiende comunicaciones y reporta desde y hacia maestro CANopen, actualiza señales de control de velocidad de servomotores y lee posición de ejes de servomotores

- Servicio *canopen\_server* implementa el protocolo de comunicaciones CANopen con tramas PDO, SDO, NMT, LSS, SYNC y *Heartbeat*.
- Servicio *pwmSinglebitport* genera señales PWM
- Servicio *do\_multi4\_qei* recibe señales en cuadratura desde codificadores y mantiene cuenta de posición

Una representación esquemática de la comunicación entre los núcleos que forman esta implementación se muestra a continuación:

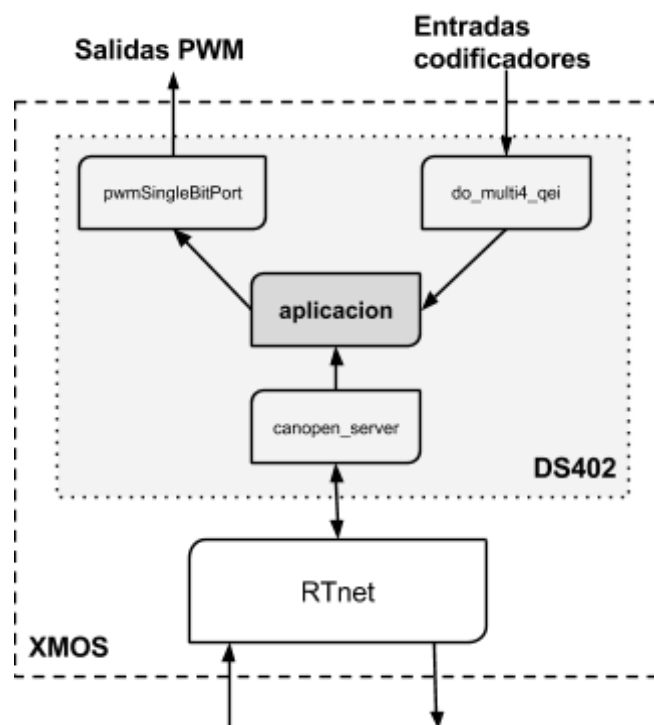


Figura 4.7: Organización de núcleos de la implementación DS-402

La organización física de las estructuras de hardware dentro del procesador XMOS es bastante simétrica en cuanto a la funcionalidad de los núcleos lógicos, en general permite seleccionar arbitrariamente cuál de todos los núcleos disponibles será encargado a una determinada tarea, pero hay excepciones. Es conveniente tener presentes las divisiones internas del procesador XMOS, en particular si se encuentra dividido en 2 *Tiles* o

“pastillas” como es el caso del utilizado en este proyecto. Cada pastilla o *tile* posee un juego de recursos y una memoria RAM independientes. Las distintas pastillas o *Tiles* componentes de un mismo procesador disponen de canales de hardware de comunicación mutua lo que permite trabajar entre dos núcleos de distintas pastillas como si se tratara de núcleos de una misma pastilla. Cuando dos rutinas ejecutando en núcleos independientes deben compartir una misma estructura en memoria RAM ambos núcleos deben ser de la misma pastilla. En el caso de este proyecto los servicios *aplicacion* y *canopen\_server* comparten un mismo Diccionario de Objetos residente en memoria RAM, como consecuencia ambos servicios deben correr en una misma pastilla.

### 4.3 Posibles mejoras a la implementación de RTnet

En la implementación actual de RTnet el arribo de las tramas Sync es cronometrado y utilizado como referencia para la recepción y transmisión de tramas y también para el procesamiento del resto de las tareas desempeñadas por el servicio *rtnet\_sync*. Esto impone limitaciones al uso del tiempo de ciclo disponible. Se puede incrementar significativamente la capacidad de carga utilizando un segundo núcleo lógico por ejemplo para el empaquetado/desempaquetado de tramas CANopen, en este caso *rtnet\_sync* recibiría las tramas RTnet ya prontas para ser transmitidas.

Además existe la limitación de transmisión de 1 trama CANopen por ciclo RTnet, esto limita significativamente el aprovechamiento de la capacidad del bus de campo. Para ello resta trabajar en la sección de chequeo del buffer de transmisión CANopen de *rtnet\_sync*.

Otra posibilidad de mejora es reducir el tiempo de activación del esclavo desde que éste es encendido. Este retardo está principalmente determinado por el proceso de calibración de tiempo de transporte que requiere de 100 ciclos de bus antes de realizar el cálculo promediado. Se podría corregir sucesivamente el tiempo de transporte recalculando cada vez que se recibe una trama de calibración.

La implementación efectuada de RTnet no cubre el protocolo de configuración RTcfg. Implementar este protocolo flexibilizaría significativamente la configuración del Bus de Campo potenciando las oportunidades de diagnóstico y servicio remotos.

### 4.4 Posibles mejoras a la implementación del Esclavo

Resulta interesante explorar la posibilidad del agregado de una segunda interfaz Ethernet aunque sólo sea para retransmitir sin procesamiento cada paquete recibido o transmitido a través de la primera interfaz. Esto potencialmente flexibilizaría las topologías de red utilizables y habilitaría la redundancia de conexión.

La placa de desarrollo XMOS utilizada mantiene capacidad ociosa para manejar 3 ejes más aparte del utilizado en la presente implementación; quedan sin uso 3 salidas PWM

con su correspondiente salida DIR y entradas para otros 3 encoders rotativos. La implementación de estos 3 ejes extra requerirá relativamente poco trabajo ya que los componentes de firmware utilizados se previeron para tal fin.

## CAPITULO 5. Plataforma de Ensayos

Con el objetivo de ensayar en la práctica al menos parte de la funcionalidad del bus de campo implementado se construyó para este proyecto una pequeña plataforma de pruebas en forma de péndulo de Furuta. Este péndulo consiste en una base de apoyo sobre la cual gira en plano horizontal un brazo cuyo extremo largo lleva adosado un segundo brazo giratorio. El primer brazo es movido por medio de un servomotor interno a la base y el segundo brazo es de giro libre. El giro de ambos brazos es detectado por medio de dos codificadores rotativos (*rotary encoder*), uno en cada eje.

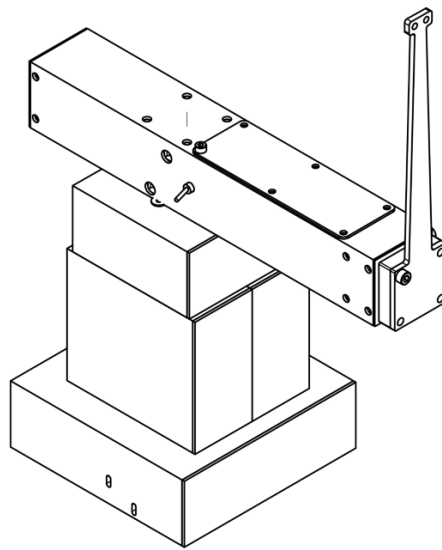


Figura 5.1: Péndulo de Furuta

El objetivo de control en el péndulo de Furuta es la orientación del brazo pasivo, el cual no tiene movimiento propio pero está sujeto a la restricción de giro perpendicular al brazo motorizado. La restricción de giro del brazo pasivo le impone una dinámica de movimiento que se debe resolver a fin de calcular los movimientos necesarios en el brazo motorizado para obtener la orientación deseada en el brazo pasivo.

El nodo esclavo implementado sobre la placa de desarrollo XMOS se conecta eléctricamente con el péndulo para obtener el control de éste desde el maestro a través del bus de campo.

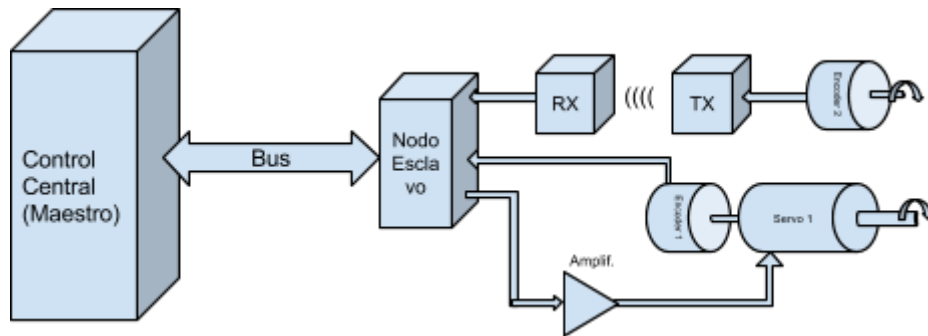


Figura 5.2: Organización de control de la Plataforma de Ensayos

El problema del control del péndulo plantea varios retos al software de control central, quizás el más simple sea el mantenimiento de una orientación vertical hacia abajo del péndulo -la figura 5.1 muestra orientación vertical hacia arriba-. Más complejo será el caso de controlar una orientación vertical hacia arriba del brazo en posición de péndulo invertido como se muestra en la figura 5.1. La solución completa al problema del mantenimiento de esta orientación incluiría alcanzar una posición de estabilidad (o pseudo estabilidad) en la orientación del brazo motorizado. El pasaje de una orientación a otra también es un problema interesante de resolver desde el sistema de control y quedan aún más variantes a introducir.

Parte del presente proyecto ha consistido en la obtención de un componente HAL para LinuxCNC que posibilite la interacción de los componentes HAL ya disponibles con el bus de campo implementado. Este componente desarrollado es apto para la conformación de sistemas de control complejos elaborados mediante la interconexión de distintos componentes HAL en el software de control central LinuxCNC.

El objetivo de ensayo en este trabajo es el de llegar a especificar el desempeño del bus de campo implementado, de manera que no se implementarán esquemas de control complejos para el péndulo de Furuta. El esquema de control del péndulo es basado en un simple lazo PID configurado para mantener la orientación vertical hacia abajo del brazo libre.

## 5.1 Hardware

La implementación del péndulo de Furuta se compone de una estructura de aluminio en tres partes: base, brazo principal y brazo secundario, con un eje de rotación entre base y brazo principal y otro eje de rotación entre el brazo principal y el brazo secundario. El eje de rotación entre base y brazo principal es propulsado mediante un motor de corriente continua y un *driver* amplificador PWM para entregar potencia a este motor; el eje de rotación entre brazo principal y brazo secundario es de giro libre. Se incorporan dos codificadores rotativos para obtener referencia de la orientación de cada uno de los dos brazos, el codificador correspondiente al brazo secundario posee un interfaz

de inalámbrico. A efectos de prevenir el volteo de la estructura frente a fallos del sistema de control se incorpora un *switch* de seguridad que detiene el giro del motor cuando la base del péndulo se levanta de la superficie.

### **5.1.1 Construcción mecánica**

El desarrollo mecánico del péndulo se realizó en software CAD lo que facilitó la integración de los distintos componentes, el diseño de las piezas estructurales de aluminio para contenerlos y las piezas de tornería para las articulaciones. Del software CAD se extrajeron modelos de cada pieza para ser importados a un software CAM en el cual se generaron los archivos de mecanizado para trabajar sobre perfiles de aluminio. Los archivos de mecanizado se utilizaron en un *router* CNC de 3 ejes para obtener las piezas estructurales que componen el péndulo a partir de perfiles de extrusión rectangular de varias medidas y de plancha de aluminio de 12mm de espesor. En el caso de las piezas de tornería para la articulación del brazo libre se obtuvieron planos directamente del software CAD para enviar a tornería. En los anexos de este documento se incluyen planos mecánicos de cada pieza componente del péndulo.

### **5.1.2 Codificadores rotativos**

Los dos codificadores rotativos utilizados entregan señal de conteo en cuadratura con 500 ciclos por revolución y señal índice (aunque su uso no está implementado en este hardware). Uno de los codificadores se instaló en el eje de rotación del brazo secundario y el otro codificador se encuentra incorporado al motor que propulsa al brazo principal.

Mediante la detección de los 4 flancos del ciclo de la señal en cuadratura se alcanza una resolución de 2000 cuentas por revolución o 0,18 grados por cuenta. En ambos ejes de rotación del péndulo ha sido implementado el conteo de los 4 flancos por ciclo para obtener la mejor resolución posible. La máxima velocidad de conteo admitida por los codificadores es de 100KHz. Esta máxima velocidad de conteo es muy superior a cualquier velocidad alcanzable por los brazos, de todas formas existirán limitaciones también en los circuitos de conteo como veremos más adelante.

### **5.1.3 Interfaz inalámbrico**

La señal de cuadratura del codificador correspondiente al brazo libre es contabilizada y transmitida al nodo esclavo por una placa interfaz diseñada

específicamente como parte de este proyecto. Esta placa consta de un microprocesador detectando los 4 flancos del ciclo de la señal del codificador y un módulo inalámbrico operando en la banda de 2.4GHz. Entre los anexos se adjunta el diseño del circuito impreso, el esquemático y el código fuente de esta placa.

El microprocesador está programado para generar un paquete de datos llevando la nueva cuenta cada vez que es detectado un movimiento en el codificador y el transmisor esté libre para transmitir un nuevo paquete. El nuevo paquete de datos es transferido a los registros internos del módulo inalámbrico por medio de un interfaz SPI para iniciar inmediatamente su transmisión finalizada la transferencia. La transmisión de datos hacia el nodo esclavo es configurada a una tasa de 1Mbps. El nodo esclavo cuenta con otro módulo inalámbrico similar para la recepción de los paquetes de datos conteniendo las cuentas del codificador.

#### **5.1.4 Driver PWM**

La potencia de giro del motor es suministrada por un amplificador driver PWM comercial especificado a 24V/20A configurado con limitación de corriente a 4A. Este amplificador recibe la señal moduladora PWM generada por el nodo esclavo para tener control sobre el voltaje medio aplicado al bobinado del motor. A su vez el voltaje aplicado influirá de forma aproximadamente lineal sobre la velocidad de giro del brazo motorizado.

El driver requiere junto con la señal moduladora PWM una segunda señal de dirección que controla la polaridad del voltaje aplicado al motor y así el sentido de giro. Además requiere alimentación con la suficiente potencia para aplicar al motor e impartir movimientos ágiles al brazo.

#### **5.1.5 Motor**

El giro del brazo principal -brazo horizontal- es obtenido mediante un motor de corriente continua integrado a la base de apoyo del péndulo, el motor está especificado en 24V/85Watts.

Este motor es también un servomotor ya que incorpora en su eje un codificador rotativo -uno de los dos que incluye el péndulo-. El motor está ubicado en el interior de la base del péndulo y su eje sobresale hacia arriba para vincularse al brazo principal.



## 5.2 Software

Para implementar el controlador del Péndulo de Furuta se requiere incorporar cierta funcionalidad específica para esta aplicación al software de control central LinuxCNC. Como se anticipaba en el capítulo 3, la configuración del software LinuxCNC para la confección de sistemas de control es extremadamente flexible. LinuxCNC dispone de una capa de abstracción muy potente en la cual se interconectan virtualmente bloques modulares de software llamados componentes HAL. Están disponibles un sinnúmero de componentes proveyendo funcionalidades requeridas comúnmente por sistemas de control de máquinas-herramienta CNC [25]. Estas funcionalidades son sumamente variadas, existen componentes tan básicos como operaciones lógicas AND, OR, XOR y los hay complejos como un generador de señal con forma de onda arbitraria, un componente generador PWM o un componente de lazo de control PID entre muchos otros. Esta característica es lo que hace aplicable la capa HAL de LinuxCNC al control de cualquier tipo de máquina, no sólo máquinas CNC.

El siguiente esquema muestra la composición del controlador implementado para ensayar el funcionamiento del bus de campo.

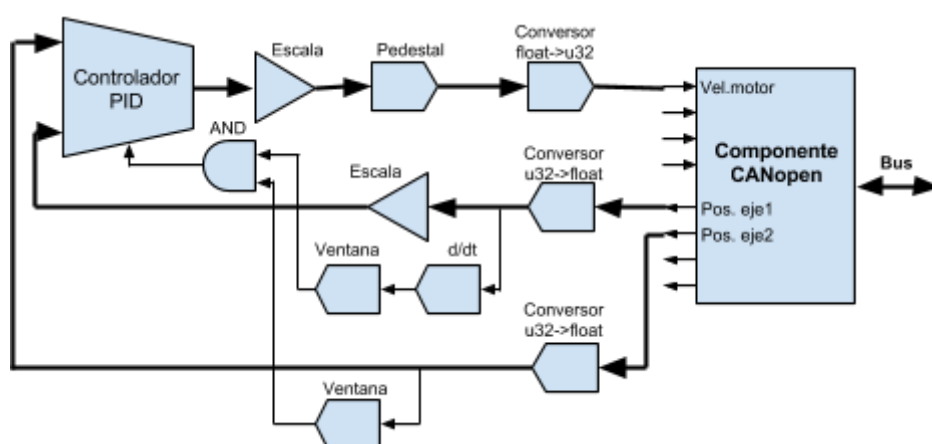


Figura 5.3. Controlador para la Plataforma de Ensayos en HAL de LinuxCNC

Para ensayar el bus de campo controlando el péndulo de Furuta hemos configurado la HAL como se aprecia en la figura 5.3 incorporando el componente fundamental CANopen desarrollado en este proyecto que provee comunicación con el bus de campo, un controlador PID estándar y circuitería accesoria para detener el giro del motor en condiciones anormales. En los anexos a este documento se detallan las fuentes del componente CANopen y del archivo HAL para la configuración utilizada.

## CAPITULO 6. Ensayos

A efectos de evaluar el desempeño del bus de campo implementado fueron efectuadas una serie de pruebas sobre su funcionamiento. Todas ellas han sido conducidas sobre la organización mostrada previamente en la figura 5.2 con un sólo nodo esclavo con su puerto Ethernet directamente conectado al puerto Ethernet del maestro a través de un cable UTP de 3 metros y sin *switches* de por medio.

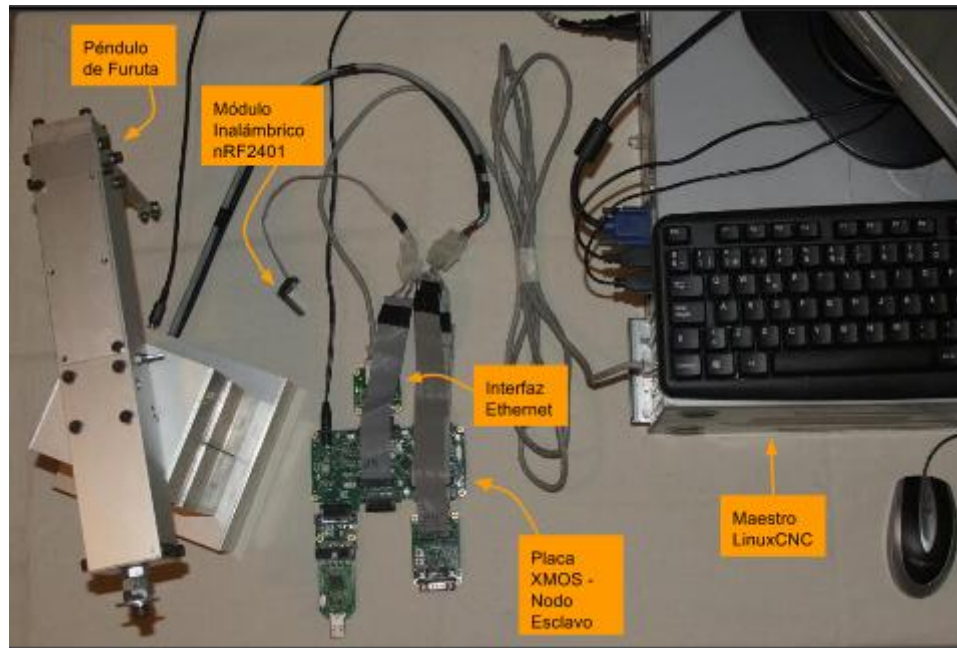


Figura 6.1: Configuración para los Ensayos.

### 6.1 Ensayos sobre RTnet

Es fundamental para el bus de campo de tiempo real el transporte rápido y consistente de tramas a través de la red, por ello las primeras pruebas realizadas sobre el bus han sido sobre el desempeño del protocolo de arbitraje de red RTnet. Se realizaron tres ensayos en este sentido: latencia, *jitter* y tiempo de ciclo.

#### 6.1.1 Latencia

Para caracterizar la latencia de transporte del bus se aprovechó el proceso de calibración del tiempo de transporte llevado a cabo automáticamente al arranque de cada nodo esclavo. El tiempo de latencia utilizado en esa calibración es una estimación basada en la simetría en las latencias de transmisión y de recepción de las interfaces, en rigor la

latencia en un sentido puede no ser igual a la del sentido opuesto; en la práctica esta estimación funciona correctamente y es el modo de calibración utilizado desde siempre por RTnet.

Para este ensayo se configuró el nodo esclavo para ejecutar su firmware en modo depuración y con un *breakpoint* al finalizar el proceso de calibración de tiempo de transporte. Alcanzado el *breakpoint* se lee el valor del registro donde se promedian 100 iteraciones de medición del tiempo de transporte. Notar que el modo depuración de ejecución del microprocesador XMOS no afecta en ningún sentido la performance de ejecución.

Este ensayo se repitió varias veces en las mismas condiciones obteniendo los siguientes resultados:

Iteración	Índice de tiempo
1	3630
2	3639
3	3643
4	3636
5	3640

Tabla 6.1: Ensayos de latencia

El valor del índice de tiempo arrojado por la rutina de calibración de tiempo en tránsito tiene unidades de 10nS y representa el tiempo de transporte de ida y vuelta de la trama de calibración. De forma que el valor tomado para el tiempo de transporte de ida con su desviación estándar es:

t.transporte	desv.est.
18200nS	25nS

El tiempo de transporte aquí obtenido representa el tiempo desde que una trama es encolada para su transmisión hasta que ésta es recibida por la interfaz del maestro, este tiempo abarca las latencias en el sistema de transmisión del esclavo, el tiempo en tránsito por el medio y la latencia en el sistema de recepción del maestro. El tiempo en tránsito por el medio físico es aproximadamente  $3 / (300 \cdot 10^6 \cdot 0.66)$  teniendo en cuenta un cable de 3m con factor de velocidad de 66%, o sea aproximadamente 15nS.

No parece prudente asumir simetría en las latencias de maestro y de esclavo de forma que sin recurrir a instrumental tendremos que tomar el valor de latencia conjunta en aproximadamente 18 micro segundos.

### 6.1.2 Jitter

Para caracterizar el *jitter* en las tramas transmitidas por el nodo esclavo se registró la actividad en el bus durante unos segundos con la herramienta de depuración de red *wireshark* filtrando exclusivamente tramas con origen en el esclavo y la trama de sincronización previa.

Los valores analizados son la diferencia entre el tiempo de emisión de la trama de sincronización y el tiempo de recepción de la respuesta desde el esclavo, esta diferencia revela la precisión y la consistencia con la que el esclavo es capaz de utilizar el intervalo de tiempo *-timeslot-* dentro del tiempo de ciclo que le es asignado.

En este ensayo el esclavo se configuró para utilizar el *timeslot* que comienza 100 micro segundos luego de la trama de sincronización. Se capturaron 100 ciclos del bus o sea 100 tramas de sincronización y 100 tramas de respuesta desde el esclavo, arrojando un retardo medio de 101 micro segundos con una desviación estándar de 1.4 micro segundos.

Dado que la desviación estándar de las medidas es del orden de la resolución de cuantificación de la herramienta de medida utilizada cabe la posibilidad de que esta desviación se reduzca si se midiera con una mayor resolución.

Jitter (rms)
1,4μS

### 6.1.3 Tiempo de ciclo

El tiempo de ciclo tiene una importante incidencia en la tasa de refresco de información alcanzable por el bus, no existirá posibilidad de actualizar un dato a intervalos menores que el tiempo de ciclo; resulta entonces importante caracterizar el bus en este sentido.

El ensayo sobre el tiempo de ciclo fué hecho con el nodo esclavo corriendo en modo depuración y comprobando el proceso de calibración de tiempo de transporte reduciendo paulatinamente el tiempo de ciclo desde 5000μS hasta 100μS. En cada iteración del ensayo fué reducido el tiempo de ciclo, reiniciado el bus y el nodo esclavo para desencadenar el proceso de calibración de tiempo de transporte. Tanto nodo maestro como nodo esclavo mantuvieron una operación normal alcanzando cada vez valores consistentes de calibración de tiempo de transporte. No se condujeron ensayos con tiempos de ciclo más cortos que 100μS ya que no resultan requeridos para el desempeño objetivo del bus. De este ensayo concluimos que la implementación del protocolo de red RTnet tanto en el nodo maestro como el nodo esclavo pueden operar correctamente a tiempos de ciclo de 100μS y también mayores.

## 6.2 Ensayos sobre CANopen/RTnet

Los ensayos sobre el protocolo de comunicación evalúan ya el desempeño del bus de campo transportando información real sobre el proceso bajo control, estos ensayos abarcan tanto la implementación del protocolo de comunicación como también el protocolo de red subyacente.

Las pruebas realizadas han sido tendientes a encontrar el mínimo intervalo de tiempo de actualización de datos sostenible por el bus de campo. Los datos intercambiados entre maestro y esclavo en estas pruebas son los requeridos para el control del Péndulo de Furuta con la organización de la fig.5.2. Los datos son: un objeto PDO (*Process-Data-Object*) de 16bits hacia el esclavo correspondiente a la velocidad de giro del motor y dos objetos PDO de 16bits hacia el maestro correspondientes a la posición de los dos ejes rotativos.

La configuración del bus de campo posee cuatro parámetros que afectan este tiempo de actualización, uno de ellos perteneciente a RTnet y tres pertenecientes a CANopen:

1. Tiempo de Ciclo (RTnet)
2. Intervalo de Sincronización
3. Intervalos de sincronización entre refrescos del Esclavo
4. Intervalos de sincronización entre refrescos del Maestro.

El intervalo de sincronización es determinado por el maestro CANopen enviando una trama específica mientras que los últimos dos parámetros establecen el número de tramas de sincronización CANopen que deben transcurrir antes que esclavo o maestro actualicen sus datos.

Estos parámetros resultan interoperantes en el bus de campo y no se pueden caracterizar por separado, de manera que se condujeron pruebas sucesivas alterando sus valores y observando el desempeño del bus.

Las tramas de sincronización CANopen son siempre emitidas por el nodo maestro, pero son utilizadas como referencia tanto por el maestro como por los esclavos, por tanto es el maestro quien regula finalmente el “caudal” de información que es transportado hacia y desde el propio maestro. De todas formas el maestro tiene limitaciones sobre el período de repetición con que puede emitir tramas *sync*, y esta limitación viene del tiempo de ciclo del bus impuesto por la configuración de RTnet. Las tramas *sync* son evidentemente “encoladas” en el ciclo del bus dentro de un slot asignado al maestro y no se puede emitir más de una trama *sync* en un mismo ciclo del bus. Aquí aparece la primera limitación: el período de repetición de *syncs* debe ser mayor o igual al tiempo de ciclo del bus. En la práctica se ha observado que la implementación del maestro requiere un período de *syncs*

cuatro veces superior al tiempo de ciclo del bus. Probablemente esto se deba a la asignación de un sólo slot para ser utilizado por el maestro y probablemente este requerimiento cambie si se le asigna más de uno.

En cada iteración ensayada con distintos valores de los parámetros del bus primero se evalúa el funcionamiento del control del péndulo -si mantiene estabilidad-, segundo se observa el intervalo de refresco de posición de los ejes rotativos para verificar si éste intervalo es regular y tercero se mide el intervalo de refresco de posición -éstos últimos con el componente HAL osciloscopio-.

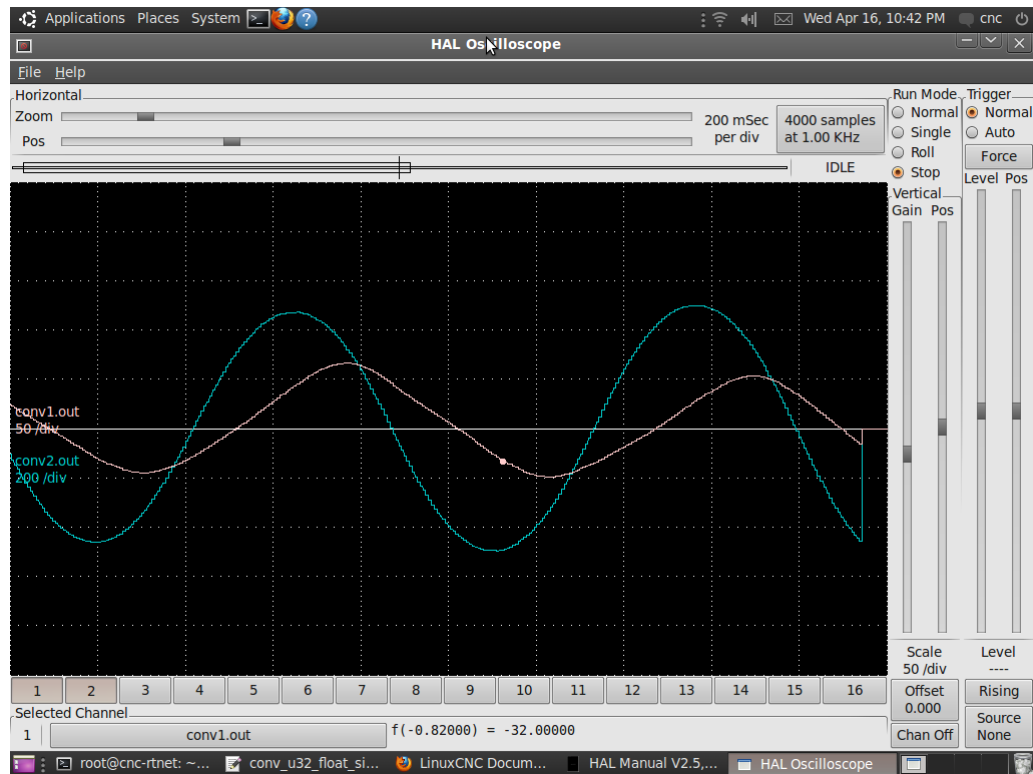


Figura 6.2: Señales de posición de los brazos en componente osciloscopio

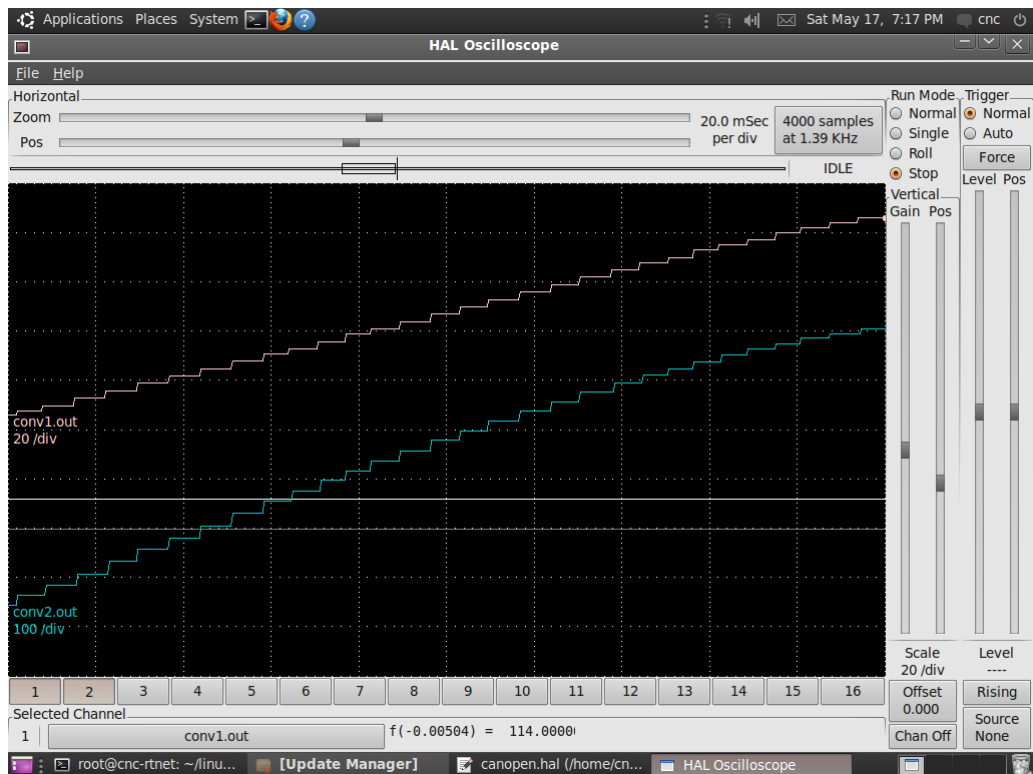


Figura 6.3: Ampliación para observar estabilidad del período de refresco.

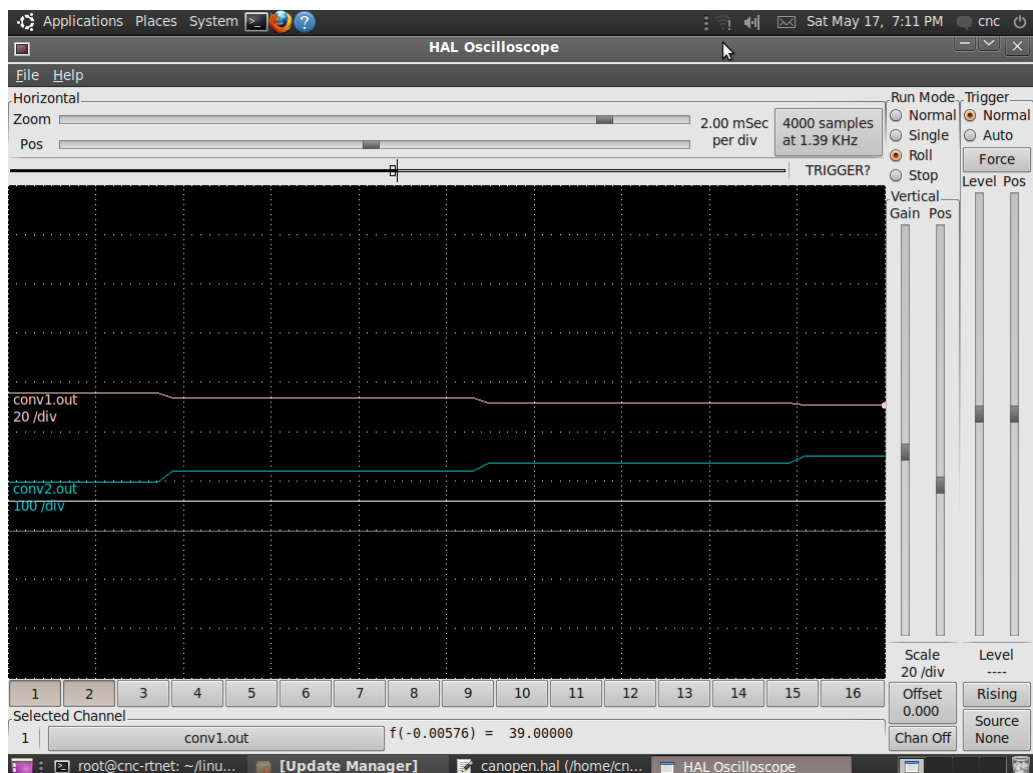


Figura 6.4: Ampliación para medir intervalo de refresco.

En la siguiente tabla 6.2 se resumen las pruebas realizadas con estos parámetros y sus resultados:

Configuración de tiempos del bus	Control	Int.Regular	Int.Refresh
T.Ciclo=1000 $\mu$ S Int.Sinc.=1000 $\mu$ S Ref.Esc.=1 sinc. Ref.Maes.=1 sinc.	No	-	-
T.Ciclo=1000 $\mu$ S Int.Sinc.=5000 $\mu$ S Ref.Esc.=1 sinc. Ref.Maes.=1 sinc.	Si	No	-
T.Ciclo=1000 $\mu$ S Int.Sinc.=5000 $\mu$ S Ref.Esc.=10 sinc. Ref.Maes.=10 sinc.	Si	Si	50mS
T.Ciclo=200 $\mu$ S Int.Sinc.=1000 $\mu$ S Ref.Esc.=10 sinc. Ref.Maes.=10 sinc.	Si	Si	10mS
T.Ciclo=200 $\mu$ S Int.Sinc.=1000 $\mu$ S Ref.Esc.=5 sinc. Ref.Maes.=5 sinc.	Si	No	-
T.Ciclo=200 $\mu$ S Int.Sinc.=800 $\mu$ S Ref.Esc.=9 sinc. Ref.Maes.=9 sinc.	Si	Si	7.2ms
T.Ciclo=100 $\mu$ S Int.Sinc.=800 $\mu$ S Ref.Esc.=9 sinc. Ref.Maes.=9 sinc.	Si	Si	3.6ms

Tabla 6.2: Ensayos de tiempos de refresco.

De la tabla 6.2 se desprende que el menor tiempo de refresco logrado fué de 3.6mS pero con un tiempo de ciclo RTnet demasiado corto para acomodar varios time-slots de 50 $\mu$ S, reduciendo la duración de los time-slots sería viable utilizar tal tiempo de ciclo. Nosotros hemos tomado la mejor configuración con tiempo de ciclo de 200 $\mu$ S previendo la incorporación de varios nodos esclavo al bus cada uno requiriendo un time-slot exclusivo. Con esta configuración el nodo esclavo alcanza un intervalo de refresco de 7.2mS para todos los datos del péndulo.

De estas pruebas se pueden realizar algunas inferencias analíticas sobre el funcionamiento de esta implementación. La segunda y la quinta iteraciones si hubieran funcionado entregarían un intervalo de refresco de 5mS, éste mismo intervalo es claramente factible de ser superado con una configuración diferente como lo muestra la



última iteración. Esto revela que no es cuestión de tiempo para que el nodo esclavo responda correctamente sino una cuestión de eventos en el bus.

Transcurriendo 5 tiempos de ciclo RTnet entre refrescos el esclavo comienza a responder, el ensayo muestra que el péndulo comienza a ser controlado. Transcurriendo 25 tiempos de ciclo el comportamiento había mejorado pero aún no era estable. Ya con 36 tiempos de ciclo transcurridos entre refrescos el esclavo mantiene un ritmo consistente de actualización bidireccional de la información.

Estudiando la solución de código desarrollado para la implementación del protocolo de red RTnet en el firmware del esclavo se encuentra justificación al comportamiento observado. Por diseño, el servicio *rtnet\_sync* ejecuta todas sus tareas en estricta sincronía con la recepción de tramas de sincronización RTnet, manteniendo los tiempos de ejecución de las tareas cortos a fin de volver a esperar la eventual llegada de una trama de sincronización. Esto hace que tareas complejas sean completadas en etapas, por ejemplo son requeridas 3 tramas de sincronización RTnet -3 etapas- para que una trama CANopen sea efectivamente entregada al proceso *canopen* para ser procesada; no importa el intervalo de tiempo entre tramas de sincronización sino el número de éstas.

## CAPITULO 7. Conclusiones

En los ensayos se probaron diferentes configuraciones de temporización del bus de campo con objeto de encontrar las limitaciones de la implementación del bus tal cual se la ha descrito. Los resultados alcanzados con estos ensayos revelan varias oportunidades de mejora en el desempeño, algunas involucran configuración del protocolo de red en el nodo maestro y otras la implementación del protocolo de red en el nodo esclavo.

Los diferentes ensayos realizados han introducido variantes en la configuración del bus desde el nodo maestro, el nodo esclavo ha permanecido intacto durante estas pruebas pero siempre sujeto a recibir la configuración impuesta desde el maestro. Los ensayos conducidos han sido orientados a la optimización de la frecuencia de actualización de la información transportada por el bus, no se han hecho optimizaciones sobre otros parámetros de desempeño del bus como *jitter* por ejemplo.

### 7.1 Latencia y *jitter*

La medida efectuada sobre latencia en el transporte de tramas entre esclavo y maestro arrojó un valor de  $18\mu\text{S}$  con un *jitter* RMS de algo más de  $1\mu\text{S}$ . Asumiendo una distribución gaussiana para los tiempos de latencia de cada trama en torno a su valor esperado, un 68.2% caerían dentro del intervalo  $\pm 1.4\mu\text{S}$  y sólo 1 trama cada 500 millones caería fuera del intervalo  $\pm 8.4\mu\text{S}$  en torno a los  $18.2\mu\text{S}$ .

Estos resultados junto con el tamaño de trama de  $6\mu\text{S}$  previamente calculado sirven para establecer un criterio en cuanto la mínima duración de los *time-slots*. Deberíamos dejar un intervalo de seguridad de al menos  $20\mu\text{S}$  entre tramas lo cual nos arroja una duración del *time-slot* de al menos  $26\mu\text{S}$ , será éste el mínimo *time-slot* que podríamos utilizar con seguridad. Se ha optado por mantener la duración de los *time-slots* en  $50\mu\text{S}$  previendo la eventual emisión de tramas de largos mayores a los 72 Bytes mínimos de Ethernet, por ejemplo al intercambiar tráfico no-prioritario entre maestro y esclavos.

### 7.2 Período de refresco

El tiempo de ciclo del bus ha sido reducido hasta  $200\mu\text{s}$  sin apreciar diferencia alguna en las condiciones de operación más que el evidente incremento de la frecuencia de actualización de la información. Tiempos de ciclo bastante menores probablemente serían tolerados por igual, pero si pretendemos mantener un tiempo de slot de  $50\mu\text{s}$  no parece razonable limitar la capacidad del bus a menos de 4 nodos esclavo con slots exclusivos para cada uno.

El período de repetición de *syncs* CANopen fué reducido hasta 800 $\mu$ s encontrando reportes de desborde de buffers en el maestro por debajo de este valor.

El período de repetición de PDOs fué posible reducirlo a 9 *syncs* CANopen, por debajo de este valor el esclavo cesa de emitir sus PDOs de forma predecible, sugiriendo también desborde de buffers pero del lado del esclavo.

En resumen se logró configurar el bus para transmitir 1 PDO desde el maestro hacia el esclavo y transmitir 2 PDOs desde el esclavo hacia el maestro cada 8ms. Este es entonces el período de tiempo entre dos sucesivos refrescos de toda la información recibida y transmitida por el software de control central LinuxCNC.

Aunque no se ha ensayado la presencia de otros nodos esclavos en el bus es previsible que el bus tolerará su coexistencia sin afectación al menos hasta un total de 3 nodos esclavos asignando 1 time-slot al maestro y otros 3 repartidos entre los nodos esclavos. Esta previsión se basa en que la limitación de desempeño surge de limitaciones en el procesamiento de los esclavos y no del maestro, cada uno de los nodos esclavos requerirá la misma intensidad de procesamiento que si se encontrara solo en el bus.

### **7.3 Desempeño en la aplicación**

Para tomar un punto de referencia de la eficacia que tendría el control central de una máquina-herramienta CNC con el período de refresco mínimo alcanzado en los ensayos podemos calcular un desplazamiento típico que ocurriría en las partes móviles de la máquina en el transcurso de este período de refresco. Para un *Router* CNC con aplicación al corte de maderas una velocidad típica de desplazamiento de las partes móviles durante el trabajo de corte de materiales es de 2000mm/min, lo que arroja un desplazamiento de 0.26mm en 8ms. Este desplazamiento está en el mismo orden de la precisión mecánica alcanzable por el equipo de corte. Para poder desempeñar una acción de control efectiva sobre los movimientos de la máquina, el software de control central debería poder realizar varias iteraciones de su lazo de control dentro de un intervalo de movimiento del orden de la precisión objetivo. Esto indica que para la aplicación pretendida el sistema entrega una frecuencia de refresco insuficiente.

### **7.4 Posibles optimizaciones**

Las limitaciones encontradas en el desempeño de velocidad del bus de campo (período de refresco) se encuentran relacionadas tanto con la implementación del nodo esclavo como también con la del nodo maestro. Probablemente la limitación en el nodo maestro sea la más fácilmente abordable y tiene relación con el error reportado de ausencia

de espacio en buffer para transmitir. Esta ausencia de espacio muy probablemente sea salvable configurando al protocolo de arbitraje RTnet para la transmisión desde el maestro en varios slots del ciclo del bus y no solamente en uno.

En cuanto a las limitaciones impuestas por la implementación del esclavo estas parecen estar relacionadas con el flujo de ejecución en el servicio *rtnet\_sync*, el cual mantiene tiempos de espera inactivos y está configurado para la emisión de una sola trama CANopen en cada ciclo de bus. Además el procesamiento de cada trama CANopen es ejecutado en varias etapas, completando una etapa en cada tiempo de ciclo. Estas son limitaciones significativas al desempeño del servicio y resulta bastante previsible un importante descenso en el período de refresco mínimo si éstas limitaciones son superadas.

Las limitaciones impuestas por el servicio *rtnet\_sync* obedecen a compromisos asumidos entre las diferentes tareas que se han volcado sobre éste. Se dió prioridad a la tarea de mantener sincronismo con las tramas de sincronización por sobre las restantes tareas. Esto obligó a mantener acotado el tiempo de procesador potencialmente dedicado al desencapsulamiento de las tramas recibidas y el encapsulamiento de las tramas a transmitir, limitándolas a una sola por cada trama de sincronización recibida. Para acometer el levantamiento de esta limitación probablemente sea conveniente la utilización de un segundo núcleo lógico por parte del servicio *rtnet\_sync*, manteniendo uno para obtener sincronización consistente como hasta ahora y agregando un segundo para atender el encapsulado, desencapsulado y posiblemente también el agendado de tramas CANopen.

## Referencias Bibliográficas.

- [1] Richard Zurawski, *The industrial communication technology handbook*. Boca Raton, FL: CRC Press, 2005.
- [2] J. Kiszka y J.R. Schwebel, “Hard real-time compliant Ethernet beyond the borders of manufacturers” *A&D Newsletter*, Octubre 2004.
- [3] J. Kiszka y B. Wagner, “RTnet – A Flexible Hard Real-Time Networking Framework” presentado en: *10th IEEE Conf. on Emerging Technologies and Factory Automation*, Catania, Italia, 2005.
- [4] Proyecto Canfestival, “página principal”, 2001. [Online] Disponible en: <http://www.canfestival.org/index.html.en>
- [5] Proyecto LinuxCNC, “página about del proyecto”, 2014. [Online] Disponible en: <http://linuxcnc.org/index.php/english/about>
- [6] XMOS Ltd., “D. May, The XMOS XS1 Architecture”, 2009. [Online] Disponible en: <https://www.xmos.com/download/public/The-XMOS-XS1-Architecture%281.0%29.pdf>
- [7] M. Rostan, “Industrial Ethernet Technologies” presentado en: *ETG Industrial Ethernet Seminar Series in Europe, Asia and North America*. Núremberg, Alemania. Ago 2011.
- [8] P. Lutz, “SERCOS III - Universal Real-Time Communication for Automation” presentado en: *Manufacturing Open Forum 2010, Summit*. Tokyo, Japan. Nov 2010.
- [9] R. Kynast, “SERCOS goes Ethernet” presentado en: *Industrial Ethernet Symposium*. Chicago, IL. Ago 2006.
- [10] G.Cena, et al. “On the Accuracy of the Distributed Clock Mechanism in EtherCAT” presentado en: *8th IEEE International Workshop on Factory Communication Systems (WFCS)*: pp 43–52. Torino, Italia. May 2010.
- [11] Ethernet POWERLINK Standardization Group, “Ethernet Power link: EPSG DS 301: Communication Profile Specification V1.1.0.”, 2008. [Online] Disponible en: [http://www.ethernet-powerlink.org/en/downloads/technical-documents/action/open-download/download/epsg-ds-301-v110-communication-profile-specification/?no\\_cache=1](http://www.ethernet-powerlink.org/en/downloads/technical-documents/action/open-download/download/epsg-ds-301-v110-communication-profile-specification/?no_cache=1)
- [12] Acromag Inc., “INTRODUCTION TO MODBUS TCP/IP”, 2005. [Online] Disponible en: [http://www.prosoft-technology.com/kb/assets/intro\\_modbustcp.pdf](http://www.prosoft-technology.com/kb/assets/intro_modbustcp.pdf)
- [13] Wireshark Foundation, “Wireshark- Network protocol analyzer, Ethercat plugin”, 2014. [Online] Disponible en: <http://wiki.wireshark.org/Protocols/ethercat>
- [14] Wireshark Foundation, “Wireshark- Network protocol analyzer, Ethernet POWERLINK plugin”, 2014. [Online] Disponible en: <http://www.wireshark.org/docs/dfref/e/epl.html>
- [15] Wireshark Foundation, “Wireshark- Network protocol analyzer, Sercos III plugin”, 2014. [Online] Disponible en: <http://www.wireshark.org/docs/dfref/s/sercosiii.html>
- [16] Wireshark Foundation, “Wireshark- Network protocol analyzer, Modbus TCP/IP plugin”, 2014. [Online] Disponible en: <http://www.wireshark.org/docs/dfref/m/mbtcp.html>
- [17] Wireshark Foundation, “Wireshark- Network protocol analyzer, RTnet plugin”, 2014. [Online] Disponible en: <http://www.erwinrol.com/wireshark/>
- [18] Automation Technology Inc., “CNC Router KL-1212”, 2014. [Online] Disponible en: <http://www.kelinginc.net/CNCmachines1212.html>

- [19] Proyecto Xenomai, “página principal”, Septiembre 2012. [Online] Disponible en: [www.xenomai.org](http://www.xenomai.org)
- [20] Proyecto RTnet, “página principal”, 2010. [Online] Disponible en: [www.rtnet.org](http://www.rtnet.org)
- [21] Proyecto LinuxCNC, “página principal”, 2014. [Online] Disponible en: [www.linuxcnc.org](http://www.linuxcnc.org)
- [22] Proyecto Xenomai, “Real-Time Driver Model”, Octubre 2013. [Online] Disponible en: [http://www.xenomai.org/documentation/trunk/html/api/group\\_\\_rtm.html](http://www.xenomai.org/documentation/trunk/html/api/group__rtm.html)
- [23] Proyecto Xenomai, “RTnet:Main”, Septiembre 2010. [Online] Disponible en: <http://www.xenomai.org/index.php/RTnet:Main>
- [24] W3C, “Raw Socket API”, Mayo 2013. [Online] Disponible en: <http://www.w3.org/TR/raw-sockets/>
- [25] Proyecto LinuxCNC, “HAL Introduction”, Febrero 2014. [Online] Disponible en: <http://linuxcnc.org/docs/html/hal/intro.html>
- [26] *CANopen application layer and communication profile, Ver.: 4.2.0*, CiA 301, Febrero 21, 2011.
- [27] Proyecto Xenomai, “Life with Adeos”, Octubre 2005. [Online] Disponible en Internet: <http://www.xenomai.org/documentation/xenomai-2.3/pdf/Life-with-Adeos-rev-B.pdf>
- [28] Proyecto Xenomai, “Latency Manual Page”, Junio 2013. [Online] Disponible en: <http://www.xenomai.org/documentation/xenomai-2.6/html/latency/index.html>
- [29] Proyecto Xenomai, “Xenomai API: rt\_task\_create”, Octubre 2013. [Online] Disponible en: [http://www.xenomai.org/documentation/trunk/html/api/group\\_\\_task.html#ga03387550693c21d0223f739570ccd992](http://www.xenomai.org/documentation/trunk/html/api/group__task.html#ga03387550693c21d0223f739570ccd992)
- [30] Institute of Systems Engineering, Real Time Systems Group, Hannover University, “TDMA Media Access Control Discipline”, 2005. [Online] Disponible en: <http://www.rts.uni-hannover.de/rtnet/lxr/source/Documentation/TDMA.spec>
- [31] Proyecto Canfestival, “The CanFestival CANopen stack manual V3.0”, 2014. [Online] Disponible en: [http://dev.automforge.net/CanFestival-3/raw-file/tip/objdictgen/doc/manual\\_en.pdf](http://dev.automforge.net/CanFestival-3/raw-file/tip/objdictgen/doc/manual_en.pdf)
- [32] Proyecto LinuxCNC, “Comp HAL Component Generator”, Mayo 2014. [Online] Disponible en: <http://linuxcnc.org/docs/html/hal/comp.html>
- [33] XMOS Ltd., “CANopen Component Documentation”, 2014. [Online] Disponible en: [https://www.xmos.com/en/support/documentation?category=xsoftip&subcategory=Networking&component=module\\_canopen&doc=XM-003376-PC/1.0.0rc0.a](https://www.xmos.com/en/support/documentation?category=xsoftip&subcategory=Networking&component=module_canopen&doc=XM-003376-PC/1.0.0rc0.a)
- [34] Kollemorgan Corp., “CAN-Bus S300/S700 Fieldbus Interface Communication Profile”, 2012. [Online] Disponible en: [http://www.kollmorgen.com/en-us/products/drives/servo/s700/\\_manuals/s300-700\\_canopen\\_communication\\_profile\\_edition-05-2012/](http://www.kollmorgen.com/en-us/products/drives/servo/s700/_manuals/s300-700_canopen_communication_profile_edition-05-2012/)
- [35] XMOS Ltd., “XMOS Layer 2 Ethernet MAC Component”, 2014. [Online] Disponible en: [https://www.xmos.com/support/documentation?category=xsoftip&subcategory=Networking&component=module\\_ethernet&doc=XM-002030-PC/2.2.4rc0.a](https://www.xmos.com/support/documentation?category=xsoftip&subcategory=Networking&component=module_ethernet&doc=XM-002030-PC/2.2.4rc0.a)

## ANEXO 1: Integración de Xenomai, RTNet y LinuxCNC

Como guía inicial se usó el siguiente artículo del wiki de linuxcnc.org, escrito por Andy Pug:

[http://wiki.linuxcnc.org/cgi-bin/wiki.pl?Mesa7i80\\_Driver\\_For\\_Linuxcnc\\_On\\_Xenomai](http://wiki.linuxcnc.org/cgi-bin/wiki.pl?Mesa7i80_Driver_For_Linuxcnc_On_Xenomai)

### **Pre-chequeos que hay que realizar:**

#### 4.2.1

1. Instalar Python 2.6
2. Agregar al root en el grupo de usuario xenomai:
  - o `$sudo adduser root xenomai`
  - o `logout` y `login`

### **Instalación de linuxCNC real time:**

Al momento de configurar la instalación del LinuxCNC, para no generar errores es necesario quitar la librería MODBUS y pasar los scripts para la configuración de Tlc y Tk, el comando sería el siguiente:

```
./configure --with-tclConfig=/usr/lib/tcl8.5/tclConfig.sh  
            --with-tkConfig=/usr/lib/tk8.5/tkConfig.sh  
            --without-libmodbus
```

### **Trouble shooting de instalación**

Estos links pueden ayudar con problemas durante la instalación de linuxcnc sobre xenomai:

<http://wiki.linuxcnc.org/cgi-bin/wiki.pl?XenomaiKernelPackages>

<http://wiki.linuxcnc.org/cgi-bin/wiki.pl?XenomaiKernel>

<http://wiki.linuxcnc.org/cgi-bin/wiki.pl?NewRTInstall>

Para el troubleshooting de Xenomai, ver los siguientes links:

<http://www.berabera.info/oldblog/lenglet/howtos/realtimelinuxhowto/>

<http://wiki.linuxcnc.org/cgi-bin/wiki.pl?XenomaiKernelPackages>

<http://www.xenomai.org/documentation/xenomai-2.6/html/TROUBLESHOOTING/>

<http://www.xenomai.org/documentation/xenomai-2.6/html/xeno-test/index.html>

## ANEXO 2: RTnet

### Instalación

Previo a la instalación del RTnet, se debe instalar el Xenomai (seguir pasos en <http://wiki.linuxcnc.org/cgi-bin/wiki.pl?XenomaiKernelPackages>).

A la fecha 28/02/2014 para poder compilar RTnet son necesarios los kernel headers. Para instalarlos dar el siguiente comando:

```
$ sudo apt-get install linux-headers-3.5.7-xenomai-2.6.2.1
```

Luego es necesario crear un link simbólico en el directorio que se va a realizar el “build” :

```
$ sudo ln -s /usr/src/linux-headers-3.5.7-xenomai-2.6.2.1/  
/lib/modules/3.5.7-xenomai-2.6.2.1/build
```

Para instalar el RTnet se debe bajar el paquete desde el github:

```
$ git clone git://rtnet.git.sourceforge.net/gitroot/rtnet/rtnet  
rtnet-0.9.13  
$ cd rtnet-0.9.13
```

### Configuración sin RTcfg y conRTcap.

La configuración se hace usando la herramienta de configuración menuconfig:

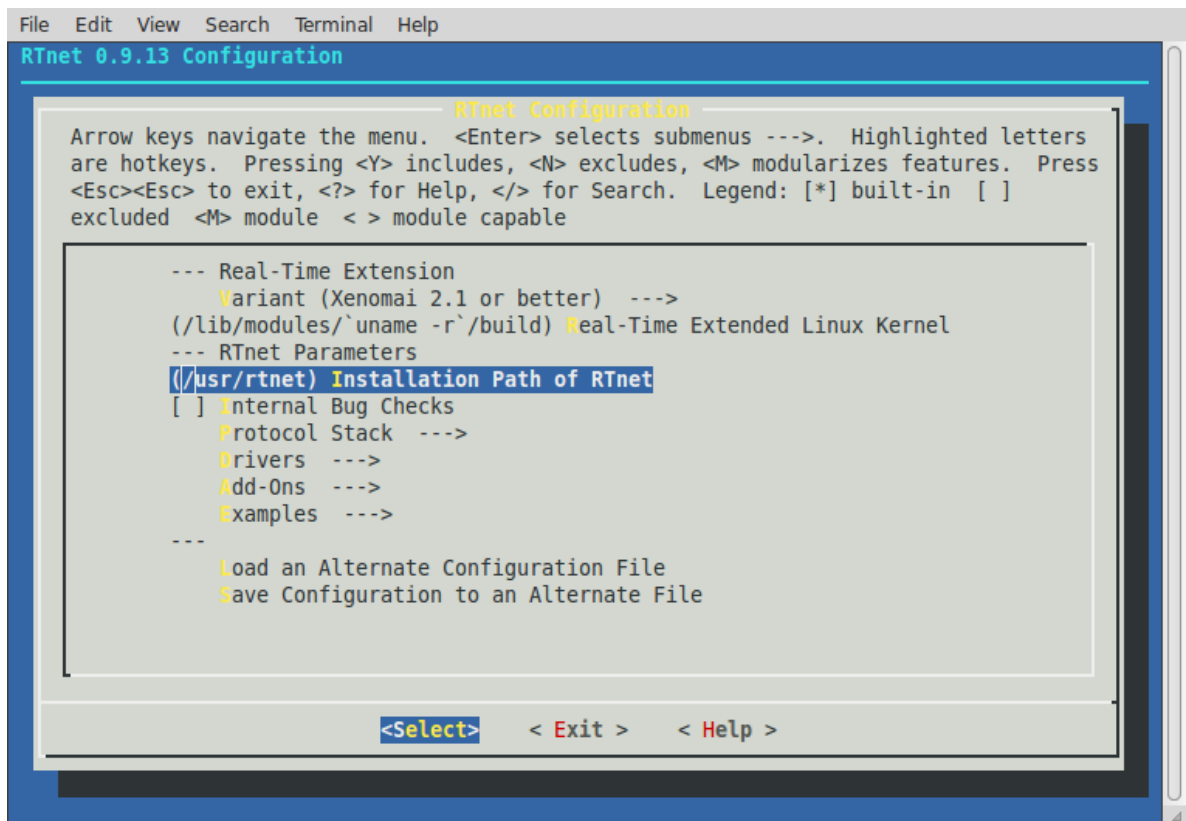
```
$ make menuconfig
```

Seleccionar las siguientes opciones:

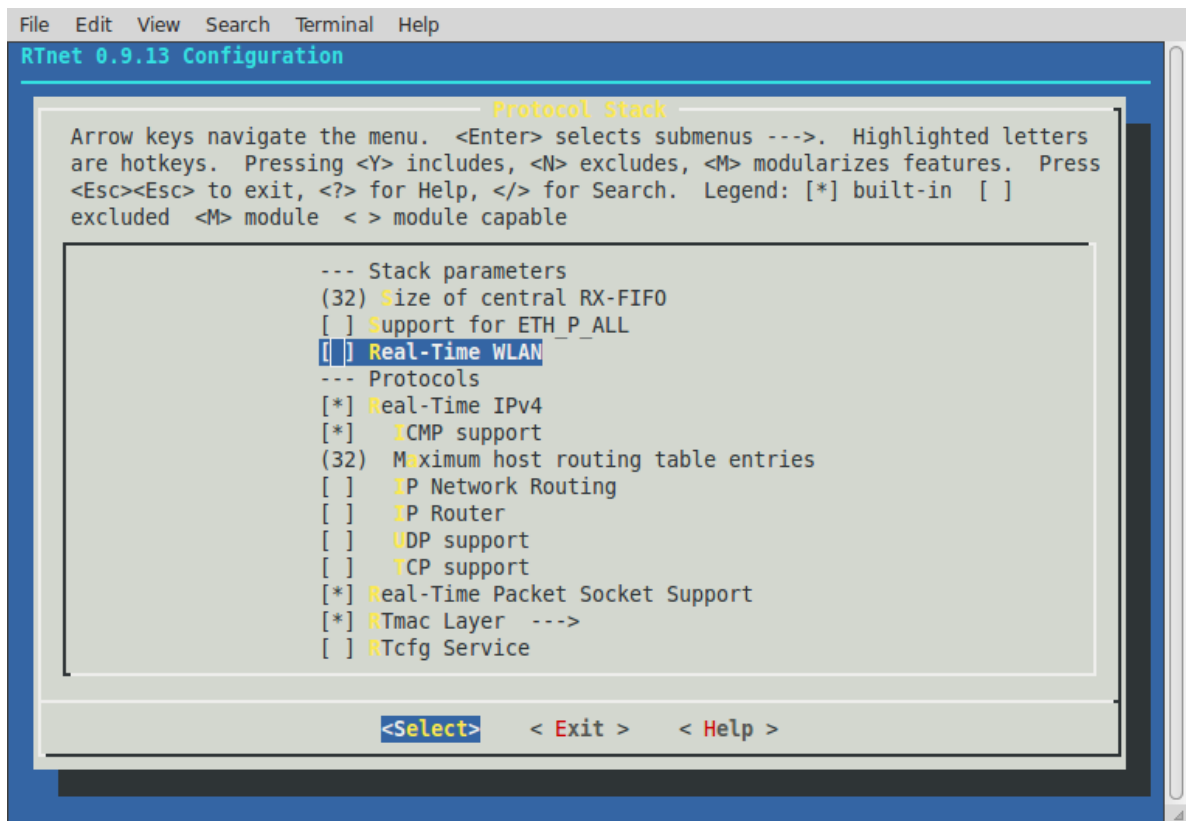
```
--- Real-Time Extension ---> Variant (Xenomai 2.1 or better)  
--- RTnet Parameters ---> (/usr/rtnet) Installation Path of
```



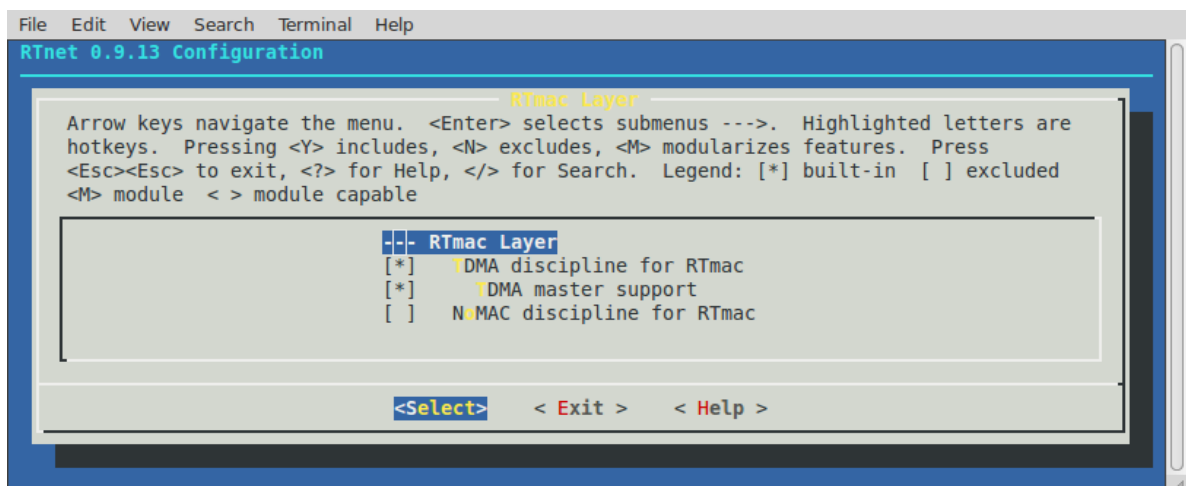
## RTnet Configuration



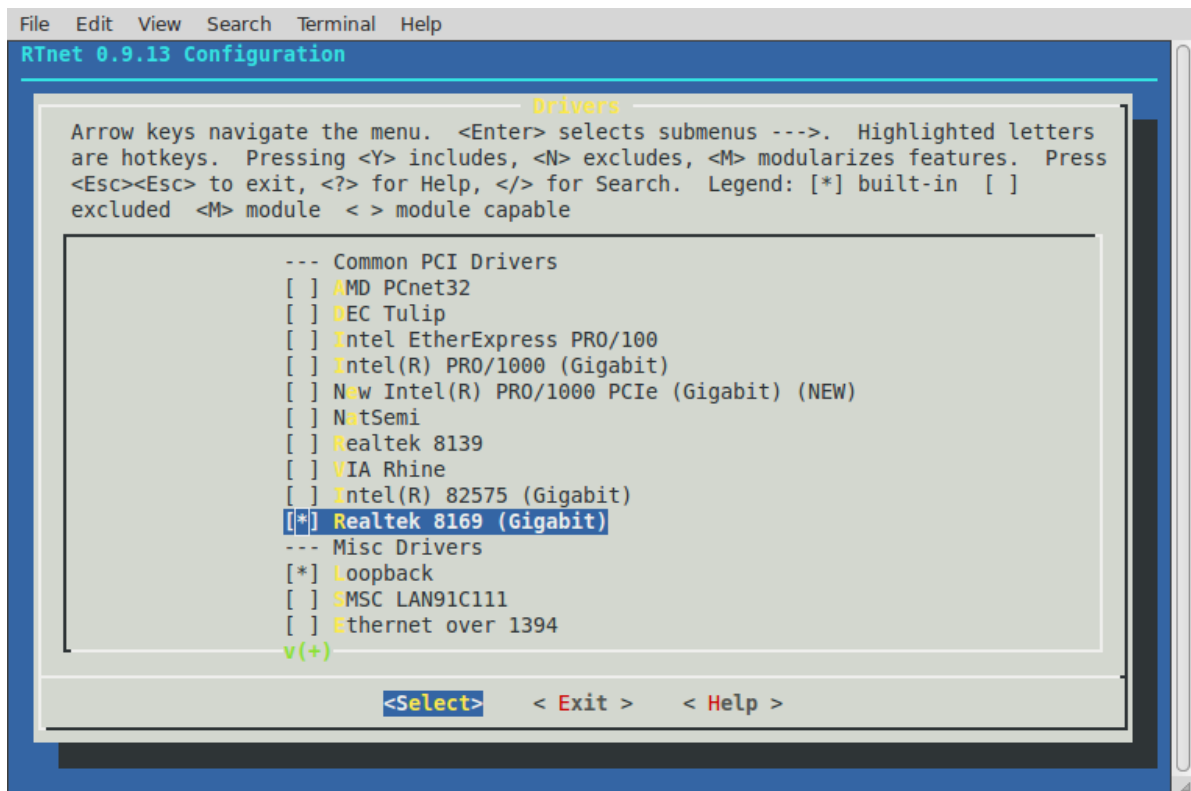
Protocol Stack --->



Se quita el RTcfg Service y el RTmac Layer se configura como sigue:

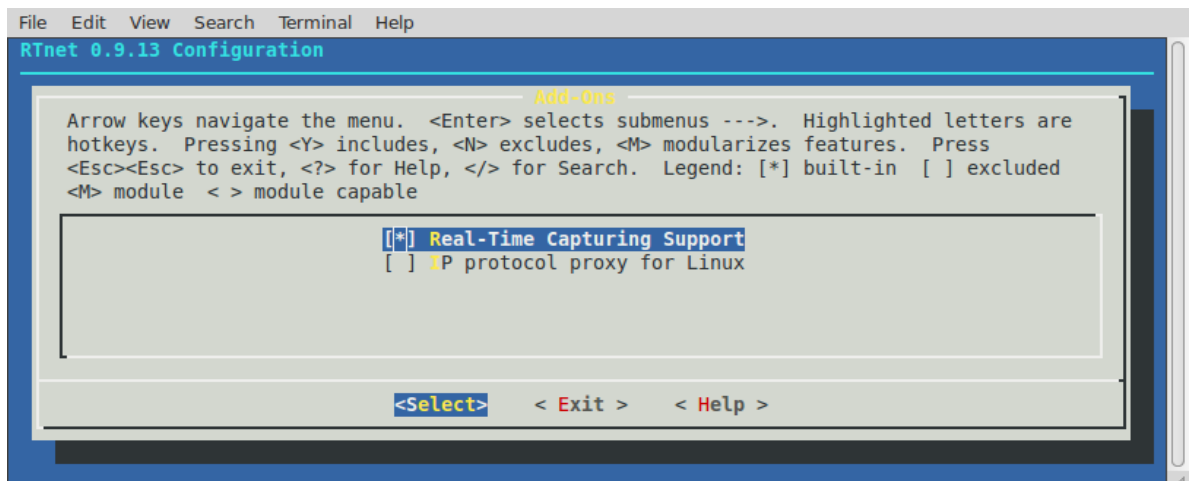


## Drivers



Se selecciona el driver de la NIC usada y el loopback.

En Add-Ons se configura RTcap para habilitar la captura con wireshark:



## **Compilación:**

Se ejecutan los comandos para la compilación:

```
$ make
$ sudo make install
```

## **Archivos de configuración rtnet.conf y tdma.conf.**

El archivo principal de configuración es el rtnet.conf, en este se debe definir:

```
...

RT_DRIVER="rt_r8169"

...

IPADDR="192.168.0.1" //Poner la ip que con la que se quiere configurar el
maestro

NETMASK="255.255.255.0"

...

RTCAP="yes" //Este flag habilita del RTcap para poder usar el wireshark

...

TDMA_MODE="master"

...

TDMA_SLAVES="192.168.0.2" // Dirección ip de o los esclavos

...

TDMA_CYCLE="1000" //ciclo de trabajo del TDM

...

TDMA_OFFSET="50"//duración del time slot
```

En caso de querer definir los esclavos a nivel de dirección MAC se debe habilitar usar el archivo tdma.conf en el rtnet.conf.

```
...

TDMA_CONFIG="${prefix}/etc/tdma.conf"
```

En el archivo `tdma.conf`, entonces se definirán los parámetros que “sobrescriben” a los definidos en el `rtnet.conf`:

```
master:

ip 192.168.3.1

mac F8:1A:67:03:E4:8F

cycle 1000

slot 0 50


backup-master: //no fue usado


slave:

ip 192.168.3.2

mac F8:1A:67:03:EB:84

slot 0 100


slave:

ip 192.168.3.3

mac F8:1A:67:03:EB:85

slot 0 150
```

### **Corriendo RTnet, levantando la interfaz rteth:**

Primero remover el driver no-realtime de la tarjeta de red (ej. Realtek 8169):

```
$sudo rmmod r8169

$sudo /usr/rtnet/sbin/rtnet start &
```

Nota: Puede dar un mensaje que no encuentra el `RTcfg` pero no es un error. Está OK sí dice “Waiting for slaves...”. Verificar que las interfaces estén arriba con “`ifconfig -a`” y listar los módulos real time en kernel con `lsmod` para ver que estén cargados.

## ANEXO 3: CanFestival

### Compilacion e instalacion

Obtener el paquete de instalación de CanFestival en <http://www.canfestival.org/code> (el lugar de descarga al 28/02/2014 es <http://dev.automforge.net/CanFestival-3/archive/a82d867e7850.tar.gz>).

Luego de descargar del paquete, se debe copiar en la carpeta .../Canfestival/drivers/, la carpeta can\_rtnet provista en el CD. Antes de compilar ver “Instalación del can\_rtnet”. La compilación de la librería con el driver can\_rtnet son tres pasos:

Configurar con Xenomai (timers=xeno) y driver can\_rtnet (can=rtnet):

```
$configure --timers=xeno --can=rtnet
```

Hacer el make y make install:

```
$make
```

```
$sudo make install
```

### Desarrollo del driver can\_rtnet

Dijimos que CanFestival tiene previsto incorporar drivers desarrollados para la interfaz que se desea. En nuestro caso fue necesario crear un driver que use el manejo de *sockets* y envío/recepción de datos según el API de RTnet, pero transportando tramas CANopen como payload.

Para esto se definió un tipo de trama Ethernet y para los campos de CANopen se usaron las estructuras de datos definidas en rtcan.h (ver [http://www.xenomai.org/documentation/trunk/html/api/rtcan\\_8h.html](http://www.xenomai.org/documentation/trunk/html/api/rtcan_8h.html) ).

### Definicion del Ethertype 0x9023 canrtnet.

El Ethertype 0x9023 fue definido como canrtnet, para el transporte de tramas CANopen sobre RTnet. Entonces, la trama será como en la siguiente

Preámbulo	Dirección Destino	Dirección Origen	Tipo	Datos	CRC
8 Bytes	6 Bytes	6 Bytes	0x9023 (2 Bytes)	<b>46 Bytes</b>	4 Bytes

En el campo “Datos” se ubica la trama can definida por rtcan.h, y tiene la siguiente forma:

COBID (can_id)	SIZE (can_dlc)	Datos
4 Bytes	1 Byte	1 a 41 Bytes

### Instalación del can\_rtnet

Tener en cuenta que la alineación de campo data[] del can\_frame de rtcan.h define el campo con alineación a 8 bytes, esto agrega 3 bytes que no se utilizan en la trama, eliminando el atributo de alineación en el archivo rtcan.h se evita esto:

Sustituir línea:

```
uint8_t data[8] __attribute__((aligned(8)));
```

por:

```
uint8_t data[8];
```

## ANEXO 4: LinuxCNC:

### HAL y sus componentes.

El Hardware Abstraction Layer del LinuxCNC, nos permite programar en lenguaje ANSI C o Python, un componente de Software que emula el un componente de hardware.

Así el componente tendrá entradas y/o salidas que serán posible conectar a con las de otros componentes.

La sintaxis del componente tiene ciertas sentencias que son interpretadas por un “compilador” que las interpreta y genera los objetos estáticos “.o” y los objetos compartidos “.so” (que pueden ser vinculados dinámicamente a un programa) para ser usados dentro del HAL. Nosotros usaremos principalmente el shell del HAL, el halcmd, aunque está incorporado a la GUI del LinuxCNC.

Para usar el compilador de componentes hay que primero correr el script rip-environment, que nos permite ejecutarlo. El comando del compilador es “comp” y hace uso del gcc con determinados argumentos.

Nota: Debe estar cargado el run in place environment (rip-environment) de lo contrario el comando comp que se usará será el del programa compare que es default de GNU/Linux y sistemas Unix en general.

Para tener un comprensión del HAL referirse al HAL Manual que es parte de la documentación de LinuxCNC.

### Compilación de componentes con comp y “manual”.

Las posibilidades del comando comp son:

- `$comp micomponente.comp // genera el “.c” micomponente.c`
- `$comp --compile micomponente.comp // genera él “.o” micomponente.o`
- `$comp --install micomponente.com // genera él “.o” y “.so”, copiando a micomponente.so a ../linuxcnc/rtlib/ para poder usarlo desde el halcmd.`

Ahora para el caso de la librería CanFestival, es necesario vincular tanto las cabeceras “.h” como los objetos. En el caso del comp, no es posible pasarle argumentos distintos, así que para un “.comp” que use la librería CanFestival, los pasos para su compilación son:

Cargar las variables de entorno:

1. `LD_LIBRARY_PATH` con los path de las lib necesarias:



- `$export LD_LIBRARY_PATH=/usr/local/lib:/home/cnc/linuxcnc-rtnet/lib:/usr/realtime-2.6.32-122-rtai/lib:/usr/lib:/lib`

## 2. C\_INCLUDE\_PATH con los path de las cabeceras necesarias:

- `$export`  
`C_INCLUDE_PATH=/usr/rtnet/include:/usr/include/xenomai:/usr/include:/usr/local/include/canfestival:/home/cnc/CanFestival/include:/home/cnc/CanFestival/include/unix:/home/cnc/CanFestival/driver/can_rtnet:/home/cnc/CanFestival/include/timers_xeno:/home/cnc/linuxcnc-rtnet/include:/home/cnc/linuxcnc-rtnet/src/hal/components`

## 3. Generar el archivo “.c”:

- `$comp micomponente.comp`

## 4. Compilar para generar el objeto binario “.o”:

- `$gcc -O2 -g -I. -I/include -I/home/cnc/CanFestival/include -I/home/cnc/CanFestival/include/unix -I/home/cnc/CanFestival/include/can_rtnet -I/home/cnc/CanFestival/include/timers_xeno -DRTAPI -D_GNU_SOURCE -Drealtime -D__MODULE__ -I/home/cnc/linuxcnc-rtnet/include -Wframe-larger-than=2560 -fPIC -I/home/cnc/linuxcnc-rtnet/src/hal/components -I/home/cnc/linuxcnc-rtnet/src/hal/components -o micomponente.o -c micomponente.c`

## 5. Vincular el “.o” con las librerías CanFestival:

- `$gcc -shared micomponente.o miDiccionarioObjetos.o -o micomponente.so -Wl,-whole-archive -lcanfestival -lcanfestival_unix -lnative -lxenomai -lpthread -lrt -lrt -ldl -Wl,-no-whole-archive`

## 6. Se copia el archivo .so al directorio rtlib:

- `$cp micomponente.so /home/cnc/linuxcnc-rtnet/rtlib/`

## **Configuración del canopen.comp desde el halcmd:**

Un script para cargar el canopen.comp en desde el halcmd se muestra como ejemplo. Modificar sus parámetros según el uso:

```
halcmd: loadrt canopen inputs=24831 outputs=24675,25600 slavenodeid=2
masternodeid=1
```

```
halcmd: loadrt threads name1=servo-thread period1=10000000
```

```
halcmd: addf canopen.0.update servo-thread
```

```
halcmd: start
```

```
halcmd: show
```

## **ANEXO 5: Hardware del Esclavo CANopen/RTnet**

### **Placa de desarrollo XMOS**

Información adjunta en archivos:

- CD/Anexo/Esclavo/XMOS-SliceKit-Hardware-Specification.pdf
- CD/Anexo/Esclavo/XMOS-XA-SK-GPIO-hwguide.pdf
- CD/Anexo/Esclavo/XMOS-XA-SK-ETH100-hwguide.pdf
- CD/Anexo/Esclavo/Esquematico\_Placa\_Madre\_XMOS.pdf
- CD/Anexo/Esclavo/Esquematico\_XMOS\_GPIO.pdf

### **Módulo inalámbrico**

Información adjunta en archivos:

- CD/Anexo/Esclavo/Hoja\_datos\_nRF24L01P.pdf
- CD/Anexo/Esclavo/Conexiones\_modulo\_nRF24L01P.pdf

## **ANEXO 6: Hardware del Péndulo de Furuta:**

### **Información Mecánica/Electromecánica**

Se adjunta en archivos:

- CD/Anexo/Furuta/Ensamble\_Mecanico\_Furuta.pdf
- CD/Anexo/Furuta/Piezas\_Mecanicas\_Furuta.pdf
- CD/Anexo/Furuta/Servo\_Linux\_85W-1.jpg
- CD/Anexo/Furuta/Servo\_Linux\_85W-2.jpg
- CD/Anexo/Furuta/Servo\_Linux\_85W-3.jpg

### **Información Eléctrica**

Se adjunta en archivos:

- CD/Anexo/Furuta/Circuito\_electrico\_Furuta.pdf
- CD/Anexo/Furuta/Interfaz\_Inalambrica\_Encoder.pdf
- CD/Anexo/Furuta/Impreso\_Interfaz\_Inalambrica.pdf
- CD/Anexo/Furuta/Codificadores\_Rotativos\_HEDS-9140.pdf

## **ANEXO 7: Fuentes de Código:**

### **Driver RTnet para CANfestival**

Se adjuntan sus archivos en: CD/Anexo/Fuentes\_Driver\_RTnet\_CANfestival

### **Componente HAL para LinuxCNC**

Se adjuntan sus archivos en: CD/Anexo/Fuentes\_Componente\_HAL\_LinuxCNC

### **Configuración de RTnet**

Se adjuntan sus archivos en: CD/Anexo/Configuracion\_RTnet

### **Implementación Esclavo CANopen/RTnet sobre XMOS**

Se adjuntan sus archivos en: CD/Anexo/Fuentes\_Esclavo\_XMOS

### **Interfaz Inalámbrica para Codificador Rotativo**

Se adjuntan sus archivos en: CD/Anexo/Fuentes\_Interfaz\_Inalambrico