Universidad Autónoma de Querétaro
Facultad de Informática

Maestría en Software Embebido

Implementar el Stack SAE-J1939 en un sistema embebido utilizando Linux.

TESIS
Que como parte de los requisitos para obtener el grado de
Maestro en Software Embebido

Presenta:
Luis Arturo López Partida

Dirigido por:
Dr. Efrén Gorrostieta Hurtado

SINODALES:

Dr. Efrén Gorrostieta Hurtado
Presidente
Firma

Dr. Jesús Carlos Pedraza Ortega
Secretario
Firma

Dr. Juan Manuel Ramos Arreguín
Vocal
Firma

Dr. José Emilio Vargas Soto
Suplente
Firma

Dr. Saúl Tovar Arriaga
Suplente
Firma

M.C. Ruth Angélica Rico Hernández
Directora de la Facultad

Dr. Irineo Torres Pacheco
Director de Investigación y Posgrado

Centro Universitario
Querétaro, Qro.
Febrero 2012
México

RESUMEN:

El éxito en el desarrollo de sistemas de software embebido implica la selección de una arquitectura que incluye primeramente el Sistema Operativo en tiempo real (RTOS(49)). Esta arquitectura puede ser propia, de código abierto y/o comercial. Cada una de estas opciones representa distintos beneficios o costos que impactan directamente al consumidor final. Dentro del mercado automotriz, en especial los sistemas de multimedia, se requiere de un gran esfuerzo por los desarrolladores para integrar tecnologías muy distintas de diversos proveedores en un solo producto. Cada OEM (fabricante de equipamiento original)(43) necesita hacer un esfuerzo independiente para integrar todas estas funcionalidades en su aplicación o RTOS especifico. Esto conlleva a un desarrollo lento y costoso, lo que ha sido una limitante para la implementación de nuevas tecnologías dentro de este mercado. Una solución a esta problemática es la estandarización de un RTOS basado en Linux y de desarrollo específico tal como lo es Android en aplicaciones de telefonía celular. Un sistema de multimedia automotriz debe cumplir con distintos requerimientos que van desde el control de suministro de energía, inmunidad al ruido electromagnético, vibración, temperatura así como la capacidad de comunicarse con los diversos buses de comunicación del vehículo como lo es el bus de CAN (Controller Area Network)(9). CAN es un protocolo de comunicaciones, desarrollado por Bosh, inicialmente para aplicaciones automotrices. Típicamente un automóvil utiliza varias redes de CAN para dar servicio a diferentes aplicaciones como lo son el motor, transmisión, multimedia y control interior. J1939 es un Estándar de Comunicación para el envío de datos a través de un bus de CAN principalmente utilizado por líneas comerciales de vehículos como lo son camiones de carga maquinaria agrícola y construcción. El protocolo J1939 provee de una comunicación uniforme entre ECUS (Unidades de Control Electrónico)(21). En esta tesis se presenta la metodología para implementar el protocolo de J1939 sobre una plataforma de Linux Embebido con el fin de ser utilizado en aplicaciones automotrices. Se utilizara un controlador de CAN comercial y sobre de esta plataforma se implementaran las capas de Enlace, de Red y de Transporte.

(**Palabras clave:** embebido, Linux, automotriz, J1939)

SUMMARY:

The success in the embedded software development depends of the appropriate architecture selection starting with the RTOS Real time Operating System. This architecture can be in-house, open source or commercial solution. Every single option offers different benefits and costs that impact directly to the final consumer product. The development of embedded software in the automotive market, especially in multimedia systems, requires an extra effort to integrate different kinds of technologies in a single product in a short period of time. This task is repeated every time the OEM selects different architectures or RTOS. As a result we have a slow development and a high cost of production in this new emerging market. As a solution, we can have a specific Software Platform for automotive multimedia devices as Android (a modified version of embedded Linux) is used in the cell phone market industry. A multimedia automotive system shall comply with several requirements or restrictions including power consumption, electromagnetic interference, temperature, vibration and the communication capability with different vehicle devices, using for example CAN (a Bosh automotive network protocol). A vehicle can use several CAN networks to control applications like motor, transmission, multimedia and internal application controls. J1939 is a protocol based on CAN and widely used in commercial heavy-duty vehicles like trucks, buses, Truck-Trailer connections and agriculture and forestry machinery. The J1939 protocol is a good solution to provide a uniform communication between ECUS (Electronic Control Units). This work presents the methodology to implement the J1939 protocol using a Linux Platform for embedded systems to be used in multimedia automotive applications. A commercial CAN controller IC, compatible with the vast majority of Embedded Linux processors, will be used to implement CAN. The Data link, Network and Transport J1939 Layers (software) will be implemented using this hardware platform.

(**Key words:** embedded, Linux, automotive, J1939)

AGRADECIMIENTOS

En la preparación de este manual se recogieron las opiniones desinteresadas de los Directores y Coordinadores de Investigación y Posgrado de la Facultad de Informática de la Universidad Autónoma de Querétaro, así como de investigadores, académicos y personal administrativo de la misma.

En particular, la Dirección de Servicios Escolares y la Dirección de Investigación y Posgrado, agradecen al Doctor Efrén Gorrostieta Hurtado el haber revisado el texto y por sus atinados comentarios para mejorarlo.

I N D I C E

INDICE DE TABLAS:

INDICE DE FIGURAS:

1. INTRODUCCION

1.1 Introducción

El uso de un RTOS en el desarrollo de sistemas embebidos ofrece grandes ventajas aun dentro del mercado de pequeñas aplicaciones. Los desarrolladores eligen el uso de un RTOS para optimizar sus tiempos de entrega aparte de simplificar el desarrollo de Software mediante modularidad y encapsulamiento. En pequeños sistemas sin RTOS se utilizan variables globales que en muchos casos son susceptibles a corromperse, siendo la causa más común de problemas. Esto se va complicando conforme se va agregando funcionalidad y en algunos casos llegando a sistemas de difícil mantenimiento. Un RTOS provee la capa abstracción para estructurar código de una manera estandarizada y simple de tal manera que se facilite su crecimiento, mantenimiento y reúso tan solo con pequeñas modificaciones en diferentes plataformas de hardware.

Las tendencias en el uso de RTOS de acuerdo con la encuesta presentada en la revista embedded.com [5] nos presenta como resultado que en 2004 más del 49% de los desarrolladores utilizaban algún tipo de RTOS. Este porcentaje creció hasta un 80.9% en 2005. También indica que existe una tendencia a preferir el uso de Código abierto, aumentando su uso de un 16% a un 19% en contraste con el Software comercial que solo aumento de un 12% a un 17%. Algunos de los factores más importantes para la selección de un RTOS son la calidad, disponibilidad del soporte y consideraciones técnicas como los requerimientos de ROM, RAM, suministro de energía y la velocidad de carga y descarga de tareas.

Dentro del mercado automotriz, en especial los sistemas de multimedia, se está requiriendo de un gran esfuerzo por los desarrolladores para integrar tecnologías muy distintas de diversos proveedores en un solo producto. Cada OEM necesita hacer un esfuerzo independiente para integrar todas estas funcionalidades o drivers en su aplicación o RTOS especifico. Esto conlleva a un desarrollo más lento y de costo elevado, lo que ha sido una limitante para la implementación de nuevas tecnologías dentro de este mercado. Una posible solución a esta problemática es la estandarización de un RTOS de desarrollo específico como lo es Android que utiliza una versión modificada del Kernel de Linux para cumplir requerimientos de la telefonía celular.

Estudio de Mercado de sistemas Embebidos 2006

Tipo de sistema Operativo Utilizado

Sistema Operativo	% 2006	% 2005
Comercial	51	44
Desarrollado internamente	21	17
Software Abierto sin soporte comercial	16	20
Distribucion comercial de software abierto	12	-

Tabla 1. Encuesta del uso de sistemas operativos [5].

Aunque el Kernel de Linux es mayor que un RTOS comercial, este ofrece la posibilidad de ser fácilmente adaptable a las diversas necesidades específicas del cliente, lo que favorece a la gran diversidad presente en los sistemas automotrices. Un sistema de audio requiere la utilización de drivers para el control de CD/DVDs, iPods, USBs, módulos de Bluetooth, GPS, tecnología G3 o incluso la implementación de protocolos de comunicación estándar en sistemas automotrices como CAN o MOST (Media Oriented Systems Transport) [11]. Estos drivers pueden ser desarrollados una sola vez y al ser de código abierto pueden ser integrados en muy diversos productos o sistemas para automóviles.

Es importante evaluar y comprender el uso de diversas tecnologías presentes en el mercado actual para dar solución de una manera eficiente y confiable para el desarrollo de sistemas embebidos de multimedia automotriz. Existen diversos tipos de RTOS: propios del desarrollador (inhouse), código abierto (Open Source) y comerciales que presentan diferentes ventajas y rendimientos a los desarrolladores; sin embargo, dada la gran variedad de tecnologías e integración de todas ellas en un solo sistema, es necesario proponer una plataforma de Software estándar. Siguiendo las últimas tendencias del mercado y por ser un código abierto “Linux embebido” puede ser una opción viable para esta estandarización [26].

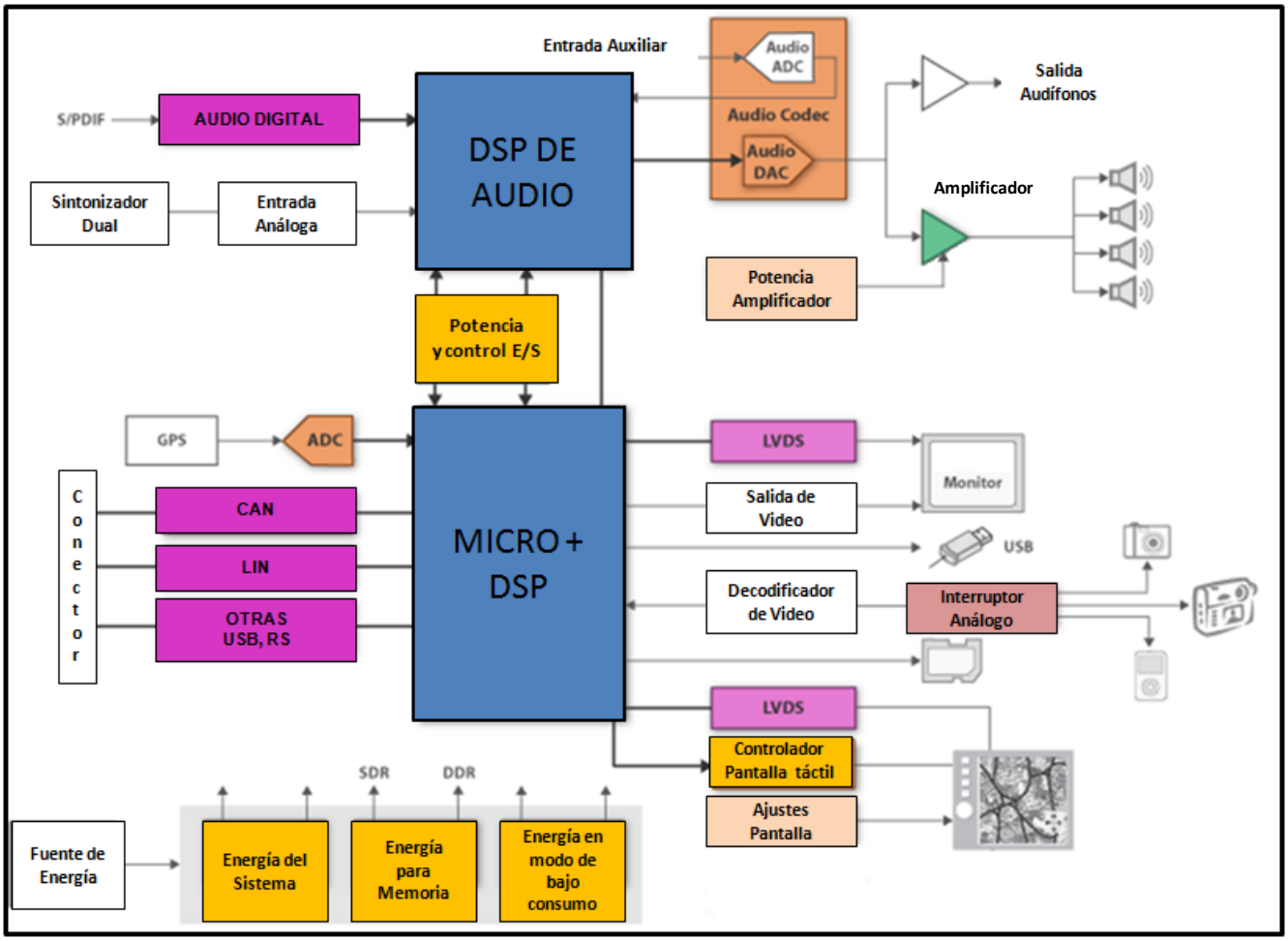


Figura 1. Implementación de sistema de Multimedia de uso automotriz [23] .

Un sistema de audio automotriz debe cumplir con variados requerimientos que van desde el control de suministro de energía, inmunidad al ruido electromagnético, vibración, temperatura así como la capacidad de comunicarse con los diversos buses de comunicación del vehículo como lo es el bus de CAN.

CAN es un protocolo de comunicaciones desarrollado por Bosh inicialmente para aplicaciones en automóviles, está basado en la topología bus para transmitir mensajes en sistemas distribuidos en tiempo real con un alto nivel de seguridad y multiplexación [21]. Típicamente un automóvil utiliza varias redes de CAN para dar servicio a diferentes funcionalidades como lo son el motor, transmisión, multimedia y control interior. CAN no posee la capacidad de transmitir audio y video dado que su velocidad máxima es tan solo 1Mbs; sin embargo, sigue siendo muy utilizado por ser económico, confiable y tolerante a

fallas. De acuerdo con el reporte "*In-Vehicle Networking Report (4Q 2004))"* de ABI Research's[1] se prevé que existirán 528 millones de nodos de CAN utilizados en automóviles en el 2010.

J1939 es un Estándar definido por SAE (Sociedad de Ingeniería Automotriz)(51) para el envío de datos a través de un bus de CAN principalmente utilizado por líneas comerciales de vehículos como lo son camiones de carga (Volvo, Freightliner, Scania, MAN AG, Renoult), tractores y maquinaria agrícola (John Deere), así como de construcción (Caterpillar). El protocolo J1939 provee de una comunicación uniforme entre ECUS soportando el principio de Plug and Play (conectar y usar).

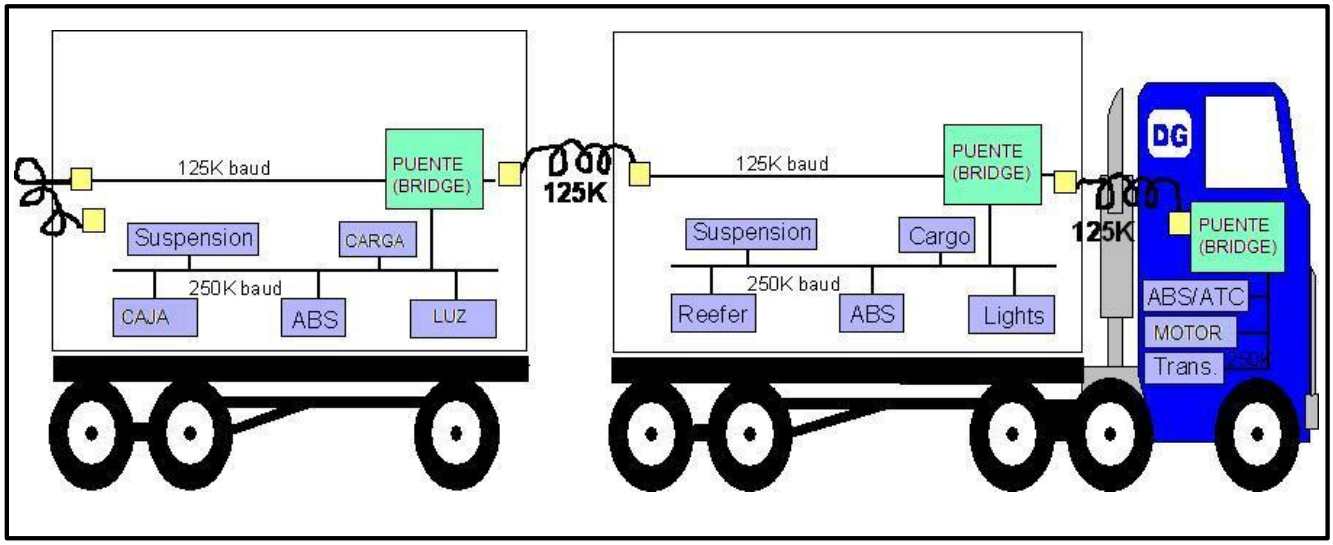


Figura 2. Implementación de protocolo J1939 en Vehículo Comercial.

1.2 Objetivos generales y específicos

Es factible y costeable la utilización de una plataforma de Código Abierto (Open Source) como Linux Embebido para desarrollar sistemas de audio automotriz. Una de las exigencias importantes de este segmento es permitir la comunicación del dispositivo con una red de CAN. En esta tesis se desarrollara el Stack de J1939 y permitirá grandes ahorros de tiempo y costos de implementación en sistemas de audio embebido.

El Stack se desarrollara en un sistema mínimo (tarjeta de desarrollo) capaz de soportar Linux. Para Implementar CAN en este sistema se fabricara una tarjeta interfaz con el controlador MCP2515/10. Dicho controlador es activado y controlado mediante SPI (Serial Peripheral Interface)(55), por lo que será necesario implementar los drivers de SPI y CAN específicos para el Hardware en el Kernel de Linux.

Para validar el desarrollo del sistema, se implementara una aplicación que interactúe con 2 módulos de CAN, un controlador de dispositivo en el volante SWC (Steering Wheel Controls)(59) o un RSA (Rear Seat Audio)(48) y un monitor de mensajes de CAN implementado en una PC.

El producto principal de esta tesis es el Building Block (bloque o unidad de construcción de software)(8) del Stack de J1939 para Linux considerando su documentación para hacerlo un producto reusable. Este proceso requiere del desarrollo de un prototipo basado en una tarjeta de desarrollo con procesador ARM así como la implementación del Hardware necesario para la interfaz con el controlador de CAN MCP2515/10 mediante SPI. El sistema implementado deberá de ser lo suficientemente robusto para cumplir con los requisitos del Estándar de J1939. El resultado de este trabajo pretende ser aplicado en proyectos embebidos de uso automotriz que comiencen a utilizar la plataforma Linux como base de su desarrollo.

El sistema podrá ser fácilmente adaptable desde el punto de vista de Hardware y Software. Podrá integrarse a cualquier equipo que soporte el puerto SPI, cubriendo de igual manera los requerimientos de ROM y RAM de la aplicación. Sin embargo, no hay que olvidar que dependiendo el caso pudiera ser necesario desarrollar un driver específico para controlar el Hardware. Una vez integrada la Capa de abstracción de Hardware, el Stack de J1939 podrá ser instalable en la plataforma de Linux embebido y será configurable de acuerdo a los requerimientos de la aplicación específica.

Para probar la funcionalidad del Stack se desarrollara una aplicación que simule la comunicación con otros dispositivos como puede ser un SWC y un monitor de mensajes de estatus. El sistema recibirá mensajes de CAN desde el SWC y la acción de cada mensaje será recibida y desplegada por la aplicación. Los mensajes de status serán enviados a un monitor de CAN. Se pretende desarrollar de igual manera la documentación del diseño, implementación y pruebas.

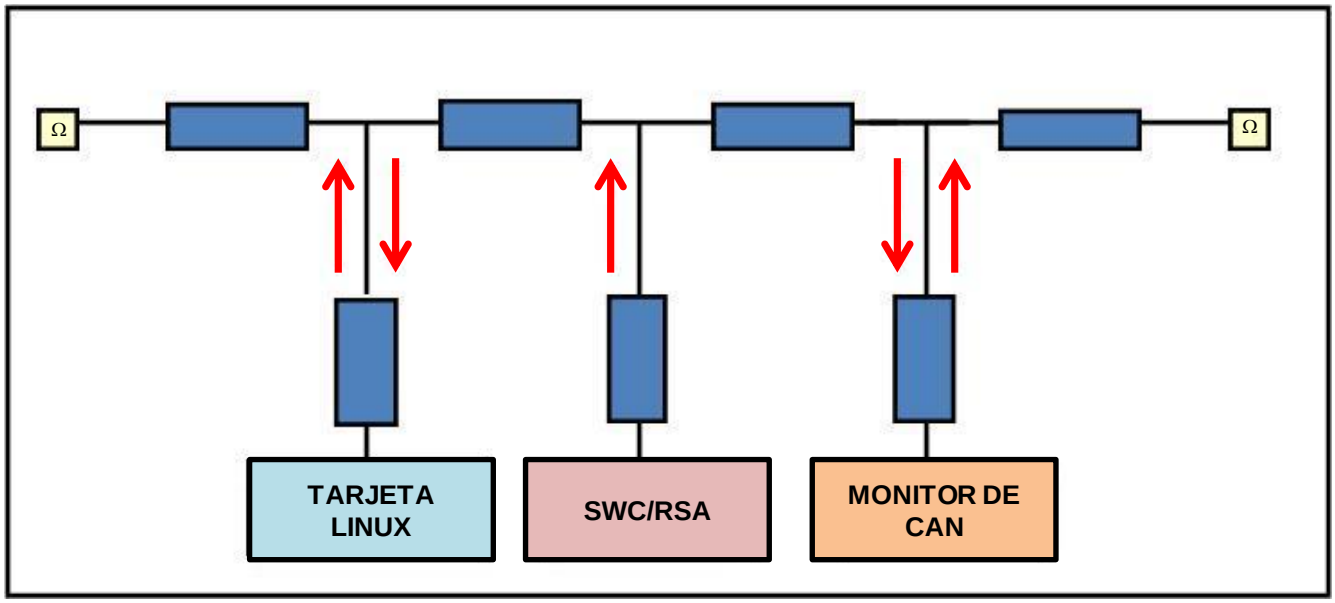


Figura 3.Flujo de mensajes en la red de CAN.

2. MARCO TEORICO

2.1 Estado del arte Hardware:

Requerimientos de Linux Embebido:

Un Kernel de Linux completo requiere de al menos 1MB de memoria, sin embargo, el Micro-Kernel de Linux actual consume solo 100K en un Pentium, incluyendo memoria virtual y todas las funciones del sistema operativo. Con el Stack de comunicaciones y utilerías básicas, un sistema Linux completo puede correr sin problema en 500K en un procesador Intel 386 con un bus de 8 bits. Simplificando su operatividad un sistema Linux embebido puede ser adaptado para requerir solo 256KB de ROM y 512KB de RAM. Sin embargo para obtener una interfaz gráfica aceptable es aun requerido 32MB de ROM y 16M de RAM.

Considerando 4 niveles de complejidad de un sistema embebido, solo una parte de ellos puede ser implementada utilizando Linux embebido:

-*32 bit CPU +MMU*: Consiste en versiones reducidas de una computadora de escritorio. Estos sistemas contienen MMU (Unidad Controladora de Memoria). Esta unidad de control permite separar los espacios de dirección de memoria protegiendo los procesos de las aplicaciones y los procesos del Kernel. Estos sistemas pueden correr Linux sin modificación.

-32-bit CPU sin MMU: Para estos equipos se requieren versiones especiales de Linux.

-8/16 bit Microcontroladores: Estos sistemas son incapaces de soportar Linux dado que el Kernel de Linux requiere de al menos 32-bits de direccionamiento de memoria.

-Hardware dedicado: Son sistemas muy pequeños que no ocupan un sistema operativo.

Los 2 primeros niveles de complejidad están tomando gran importancia en el mercado actual dado que el costo de los procesadores continúa disminuyendo y esto implica poder utilizar el Software existente con un mínimo de modificaciones.

El uso de MMU permite compartir librerías dado que el mismo segmento de código puede ser mapeado a múltiples procesos. Esto reduce grandemente el procesamiento en la memoria de trabajo así como en el sistema de archivos. Entre más grande y compleja sea la plataforma más importante es compartir para reducir los requerimientos de memoria.

Existe una gran tendencia por utilizar soluciones SOC (Sistema en un integrado)(53). Actualmente, la RAM y ROM continua estando dentro de los mismos circuitos integrados. Los SOC's integran un CPU de propósito general y la gran mayoría de los controladores de entrada/salida presentes en la tarjeta madre de una PC estándar. Las más importantes arquitecturas son:

PowerPC: Es una arquitectura típica RISC(48) de 32 bits de propósito general. Incluye 32 registros de 32 bits, MMU y caches. Existen muchos SOC's como el MPC623e que implementan la arquitectura PowerPc con MMU, un DSP(19) de 16 bits, hardware dedicado con más de 10 protocolos de comunicación incluyendo USB, Ethernet, ISDN(32), DMA (Direct Memory Access)(16), controlador de memoria de 8 bancos, controlador de video y LCD configurable para gran variedad de pantallas a 75Mhz de 0.2 a 0.3W de consumo.

Arm: Es una arquitectura específicamente diseñada para embebidos de bajo costo y bajo consumo de energía. Es una arquitectura RISC de 32bits con 16 registros. No es idónea para implementaciones de alto procesamiento pero funciona muy bien en embebidos económicos. Esta arquitectura reduce el ancho de banda hasta un 30%.

x86(64): Los embebidos x86 poseen menor capacidad comparados con los RISC debido a su alto consumo de energía, aunque los sistemas Linux son más maduros utilizando este tipo de arquitectura.

Procesador seleccionado:

OMAP Generación 3 de Texas Instruments:

Ventajas de la Arquitectura:

La arquitectura multiprocesador OMAP ha sido optimizada para soportar aplicaciones pesadas de multimedia tales como audio y video. Esta compleja arquitectura combina 2 procesadores heterogéneos (RISC y DSP), diversas combinaciones de sistemas operativos y aplicaciones corriendo en el DSP y el ARM son accesibles a desarrolladores de aplicación gracias la característica del puente de datos DSP/BIOS(4) . El ARM con arquitectura RISC es apropiado para el código de control: Sistema operativo, Interfaz de Usuario y aplicaciones de sistema, el DSP es apropiado para aplicaciones de procesamiento de señales como MPEG4, reconocimiento de voz y ejecución de audio.

El procesamiento típico de un Procesador StrongArm requiere de al menos 3 ciclos más de lo que requiere un DSP. De manera similar el costo de energía es al menos de la mitad utilizando un DSP.

La arquitectura DUAL CORE(20) del OMAP permite realizar una aplicación de videoconferencia con solo el 40% de los recursos por lo que resta un 60% para correr otras aplicaciones al mismo tiempo (word, excel, etc). Con un procesador RISC todos los recursos serían destinados a la videoconferencia.

Cómo funciona la Arquitectura:

Ambos Procesadores (ARM y DSP) utilizan cache de instrucciones para reducir el porcentaje de acceso a memoria y eliminar el consumo de potencia. El OMAP contiene 2 interfaces de memoria externa y un puerto de memoria interna. Las interfaces de memoria externa soportan comunicación síncrona DRAMs hasta de 100Mhz y memorias estándar asíncronas como SRAM, FLASH. El puerto de memoria interna permite dirección directa al chip como SRAM y puede ser usada para accesos frecuentes en las rutinas críticas de despliegue de datos para pantallas LCD. Todas estas interfaces son independientes y permiten accesos simultáneos desde el procesador o de la unidad del DMA.

El OMAP contiene numerosas interfaces para conectar periféricos externos desde el DSP o el ARM. Para mejorar eficiencia estas interfaces soportan DMA para cada procesador. El bus

local de las interfaces es de alta velocidad bidireccional y multimaestro. Adicionalmente un bus de alta velocidad de acceso está disponible para permitir a dispositivos externos compartir la memoria de sistema (SDRAM,FLASH, memoria interna RAM). Para soportar requerimientos del sistema operativo el OMAP incluye periféricos como: temporizadores, interfaz de entrada/salida, UART, watchdog(63), etc. El DMA del ARM contiene un canal dedicado usado para transferir datos desde el buffer de datos del controlador de LCD que puede ser alojado en la SDRAM o SRAM interna.

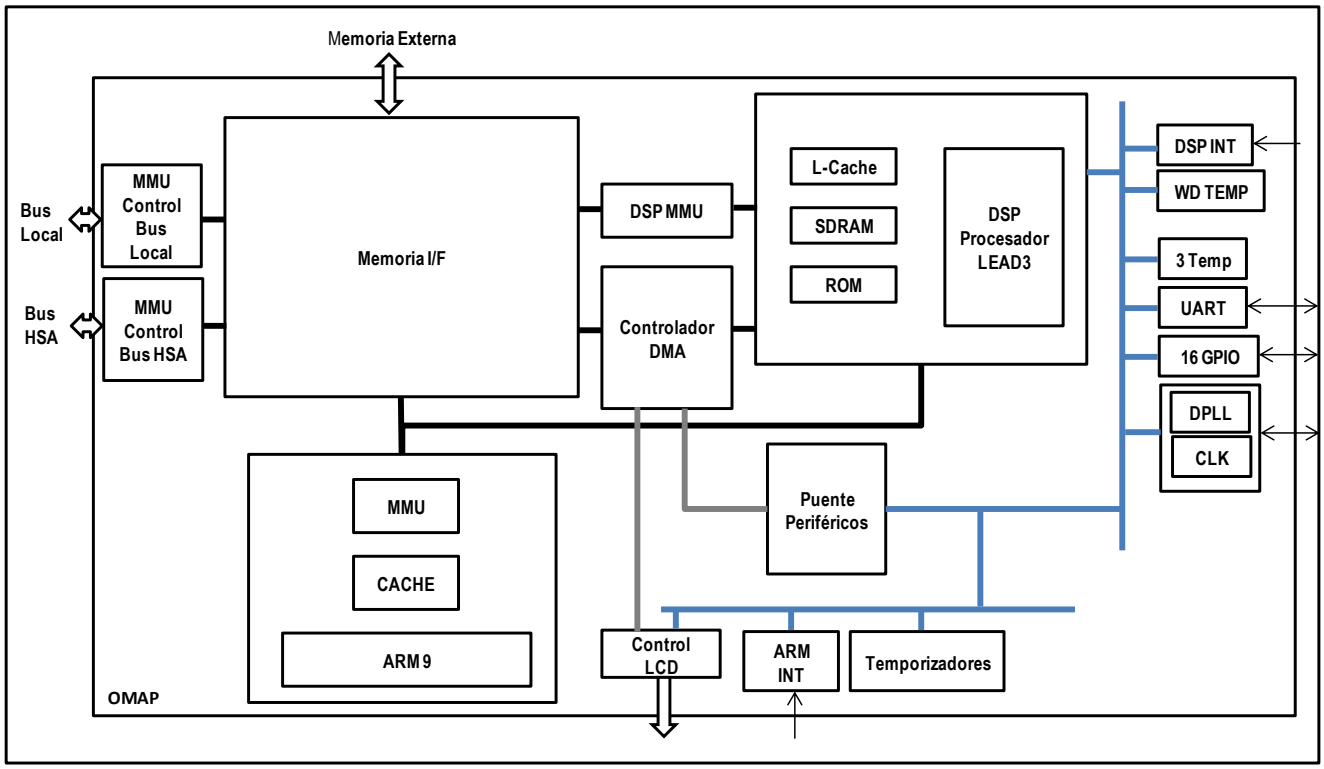


Figura 4.Estructura General de OMAP generación 3 [25].

Descripción de la arquitectura del OMAP3530:

Su arquitectura fue diseñada para proveer el mejor video en su clase y procesamiento gráfico soportando:

-Video Streaming

-Juegos en 3D

-Video conferencia

-Imágenes de alta resolución

Este dispositivo soporta sistemas operativos de alto nivel: Linux y Windows CE.

Los siguientes subsistemas son parte del OMAP3530:

-Unidad de Microprocesador (MPU) basado en el ARM Cortex A8.

-Subsistema IVA2.2 con un DSP C64x+.

-Subsistema Power SGX para aceleración de gráficos en 3D y soporte de juegos de video.

-Procesador de Imágenes que soporta múltiples formatos e interfaces para conectar una gran variedad de sensores de video (cámaras).

-Subsistema de imágenes que soporta gran variedad de salidas de video como NTSC/PAL.

-Buses Nivel 3(L3) y Nivel 4(L4) que proveen de alto ancho de banda para transmisiones de datos interna y externamente entre periféricos.

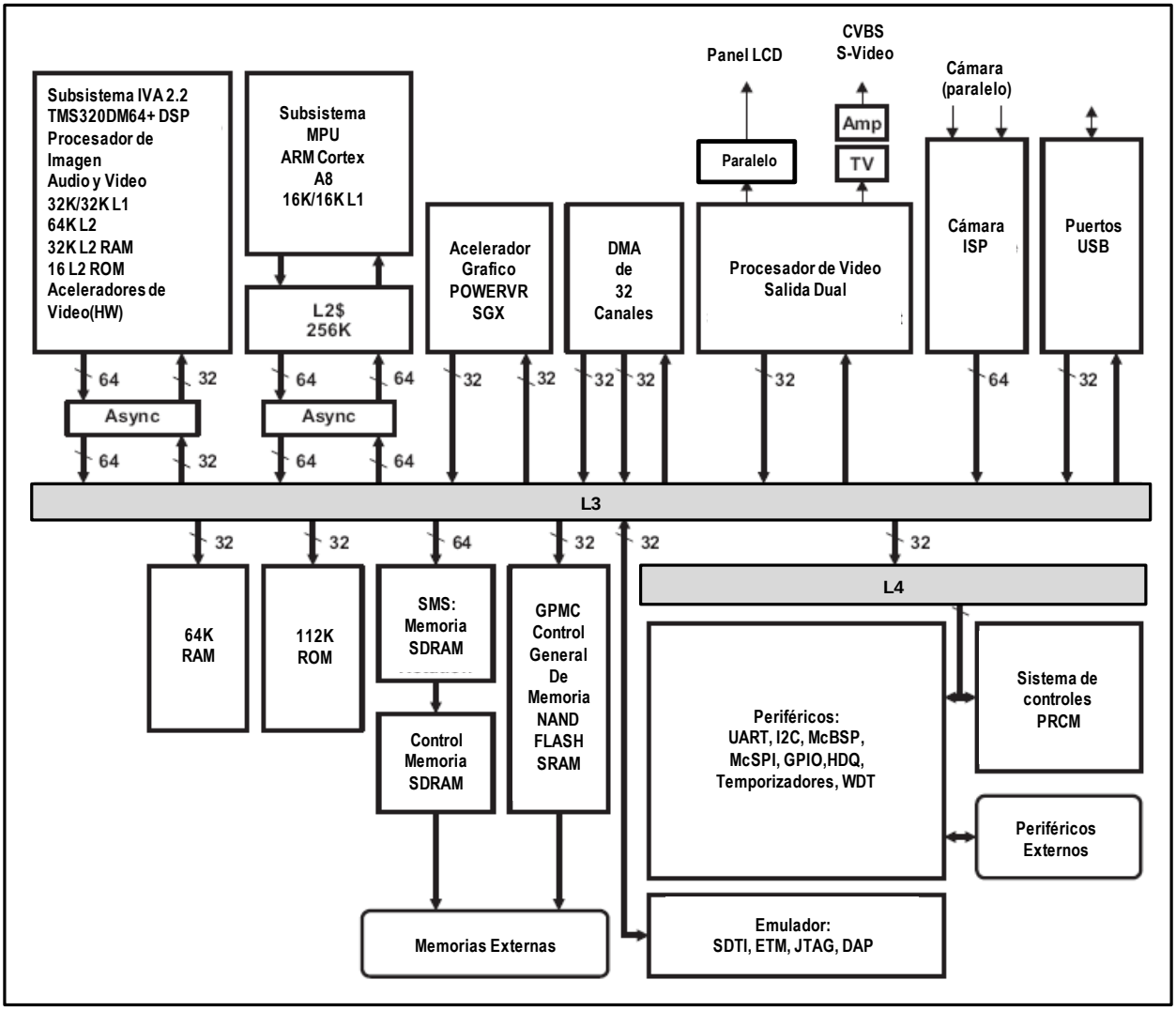


Figura 5. Diagrama de Bloques de OMAP3530 [25].

Selección de Tarjeta de desarrollo:

Se seleccionó la Tarjeta BeagleBoard por ser de bajo costo, sin requerir sistema de ventilación y ser un CPU de una sola tarjeta. Esta tarjeta incluye una gran variedad de software abierto y herramientas de desarrollo gratuitas por lo que su costo disminuye considerablemente al no requerir compilador y/o herramientas de programación.

El sistema incluye:
-OMAP3530:1200 DMIPS(17), basado en ARM Cortex-A8 a 600Mhz,TMS320C64x+ DSP a 430MHz.
-Memorias: 256MB NAND Flash y 256MB de SDRAM.
-PMIC TPS65950: IC que provee servicios al OMAP como: Reguladores de Voltaje, Audio CODECs(11),
-TFP410: Interfaz de video para diversas pantallas de despliegue.
-SD/MMC Card slot(52), que puede ser usado para contener el sistema operativo.
-Puerto USB al que se le puede conectar un adaptador Ethernet , teclado, mouse, etc
-DVI-D de salida y S-VIDEO (NTSC/PAL) .
-Audio de salida Estéreo.
-Expansión para puertos: I2C, SPI, UART, GPIO, SD/MMC
-Consumo 2W a 5V.
-Soporta versiones de Linux como: Android, Angstrom, Ubuntu, Gentoo, Maemo, etc.
-Y otros sistemas operativos como: Windows CE, Symbian, QNX, etc.

Comparación de Implementación del Hardware con FPGAs(23):

Existen algunos Soft Cores(54) que soportan versiones de Linux embebido como lo son el NIOSII de Altera [2] y el MicroBlaze de Xilinx [29]. Ambos procesadores basados en arquitecturas RISC Harvard de 32bits con MMU y pueden ser programados en un FPGA. La ventaja de estos Soft Cores es que pueden ser adaptados u optimizados a las distintas necesidades del cliente por lo que se acelera el ciclo de desarrollo del Hardware del sistema.

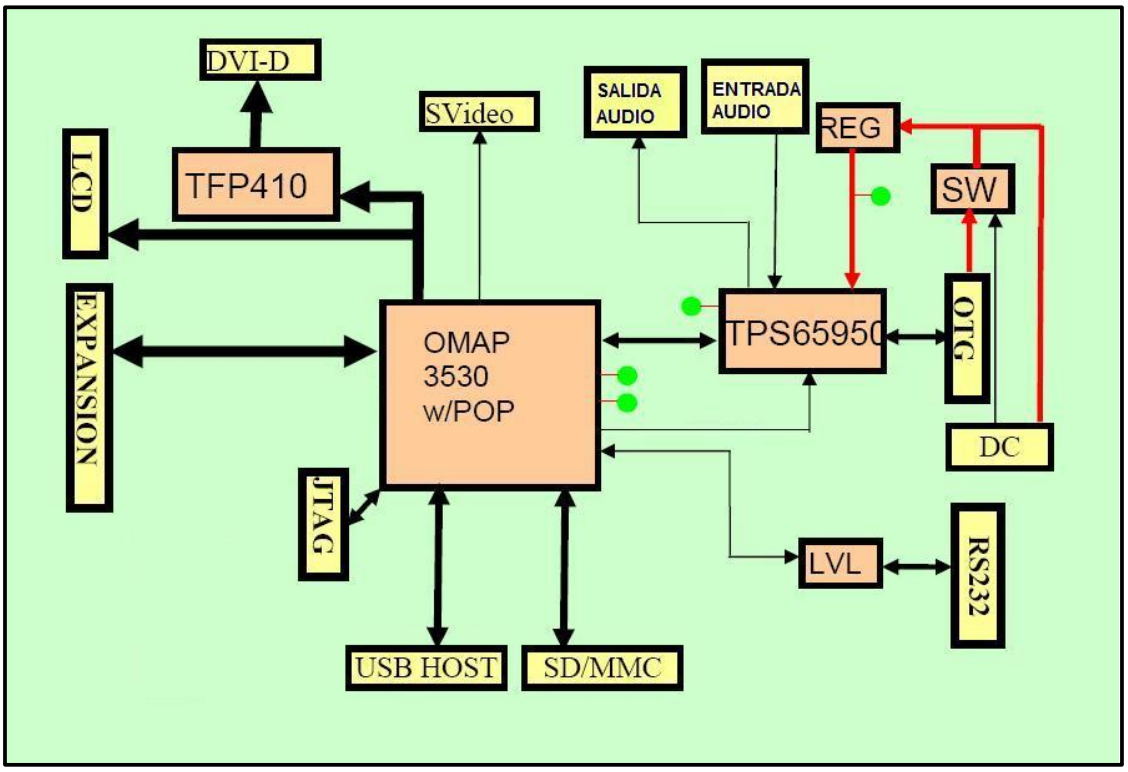


Figura 6. Diagrama de Bloques de tarjeta BeagleBoard [4].

El kit de desarrollo de Altera Nios II[2] incluye: la tarjeta cycloneIII_3c25_niosII_video, el procesador núcleo de Nios II/f es de 113 DMIPS a 100 MHz y soporta LCD's, conector JTAG, Flash de 16MBytes y SDRAM de 32MBytes, interfaces de comunicación Ethernet, UART, etc. También Ofrece la versión uCLinux como RTOS.

El procesador NIOSII/f ofrece el máximo rendimiento a expensas del tamaño ocupado en el FPGA. Incluye: Caches separadas parea instrucciones y datos (512B y 64KB), MMU opcional (no opcional para Linux), acceso de hasta 2GB de memoria externa, hasta 256 instrucciones y aceleradores de HW, JTAG para detección de fallas.

Es posible utilizar las interfaces de comunicación SPI de ambas tarjetas (Xilinx y Altera) para implementar CAN con el controlador MCP2515 de Microchip y proveer una solución similar a la de la tarjeta BeagleBoard propuesta para el desarrollo de esta tesis.

Pero también podemos implementar el Bus de CAN en un FPGA. Existen algunas opciones comerciales (CAST Inc) y su costo incluye los pagos de derechos de licencia de uso de CAN 2.0 que exige BOSH, puede ser adaptado a diversas arquitecturas de ALTERA, XILINX y otros. La implementación de este bus de CAN incluye 3 buffers(7) de transmisión y recepción. El controlador puede escribir y leer de la memoria del módulo como si fuera una RAM convencional. La interfaz del controlador es intercambiable, Todos los eventos de RX, TX y control generan interrupciones en el controlador. El controlador contiene 27 filtros (mascaras) de entrada y un Bit-Rate(5) programable de hasta 1Mbps.

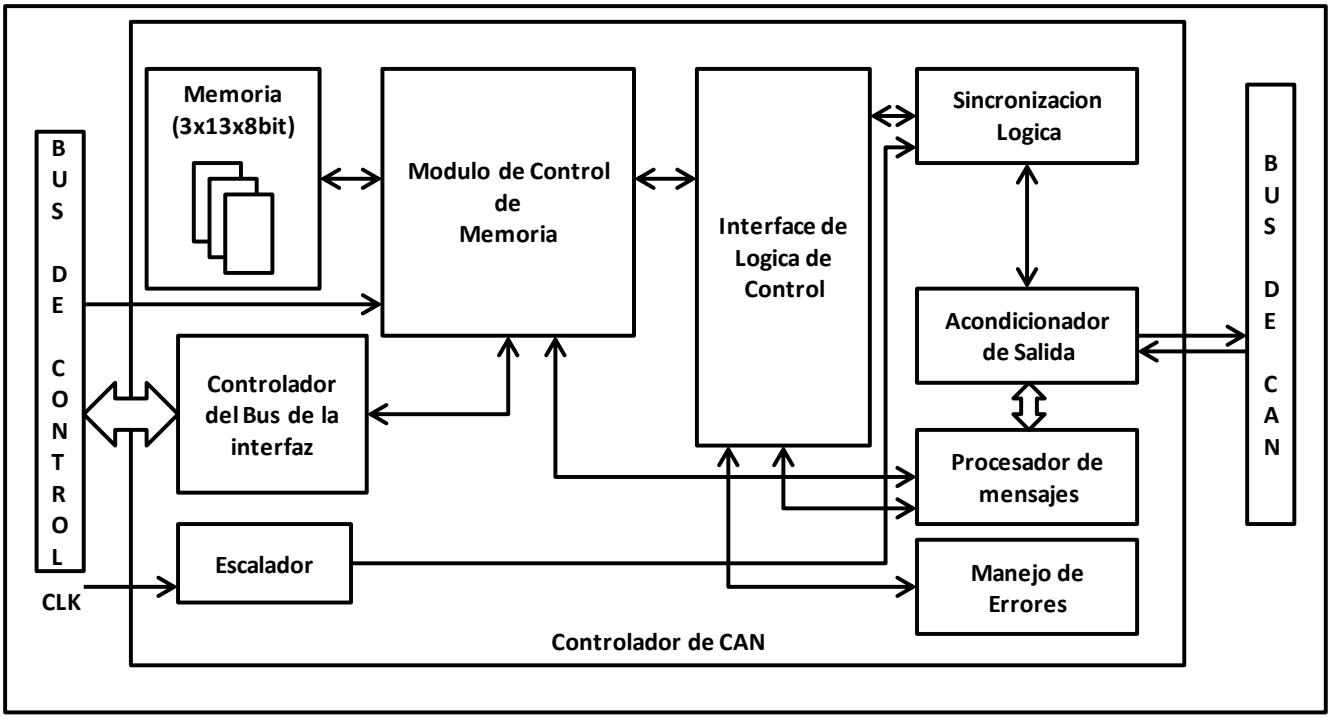


Figura 7. Controlador de CAN en un FPGA de CAST Inc [8].

Una vez integrado CAN a nuestra solución FPGA podemos implementar el desarrollo del protocolo de J1939 en el espacio de usuario de Linux. Otra solución, considerando el pago de licencias, puede ser desarrollar nuestro propio Núcleo de CAN y/o el protocolo de J1939.

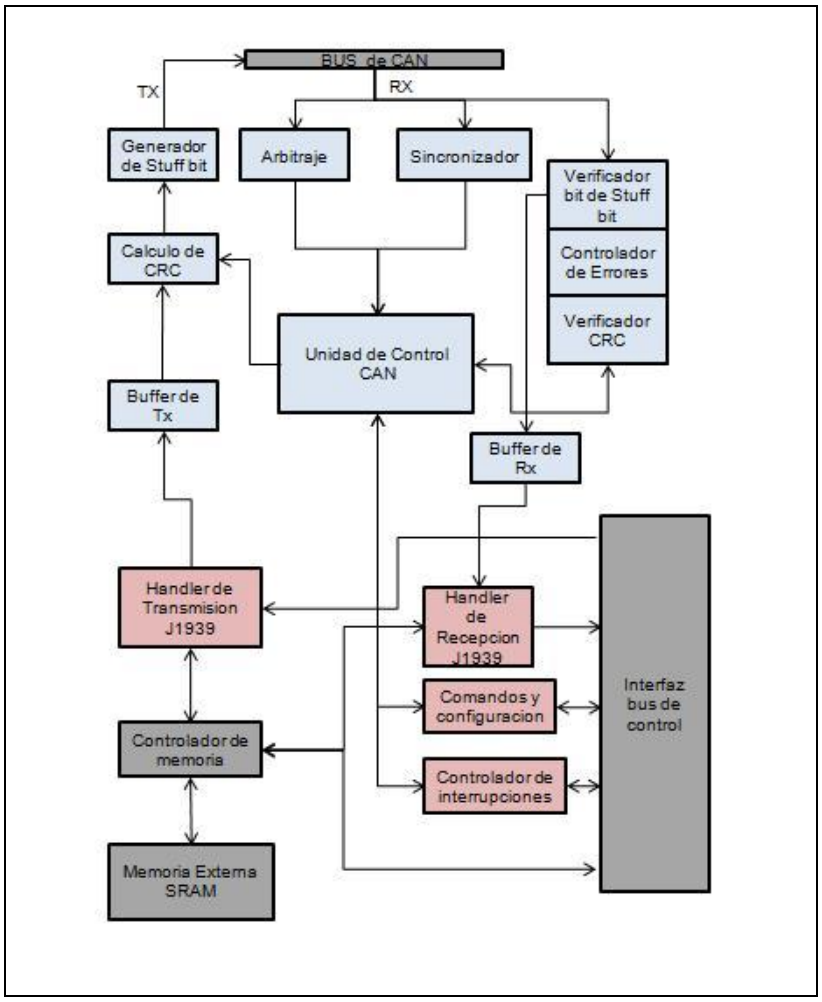


Figura 8. Implementación de Protocolo J1939 en un FPGA

Controlador de CAN:

Recepción:

Cuando el sincronizador detecta la transición de recesivo a dominante notifica al controlador de CAN que la recepción ha comenzado. Cuando el mensaje está llegando es procesado por el verificador de bit de Stuffing(58), si existe error se le notifica al controlador de CAN. El controlador de errores y de CRC (Comprobación de redundancia Cíclica)(12) informa si existe algún error en la trama de datos en la unidad de control de CAN. En caso de no existir errores, la trama de datos es transferida al buffer RX.

Transmisión:

El módulo de arbitraje detecta si el mensaje puede ser mandado cuando el bus se encuentra en Idle (Inactividad)(30) y se lo comunica a la unidad de control de CAN. La unidad de control inicia el proceso del cálculo del CRC, una vez terminado se comienza el armado de la trama de datos con los bits de stuff(57) para después proceder a su envío.

J1939:

El handler(27) de Recepción filtra los mensajes recibidos y manda los datos de la trama a la memoria externa mediante el controlador de memoria. Este controlador también identifica los PGN's(45) de J1939 para distinguir entre un mensaje o un grupo de mensajes (TP(61)).

El handler de Transmisión arma los PGNS con los datos a enviar tomando los datos de la memoria externa, el bus de control previamente escribe los datos en la memoria antes de ser leídos por este handler.

Conclusiones para la selección del Hardware:

La siguiente tabla comparativa muestra la diferencia entre los 3 sistemas:

Kit de desarrollo:	**BEAGLE BOARD**	**Cyclone III**	**Virtex-6**
Procesador	OMAP3530	NiosII	Microblaze
Marca	TI	Altera	Xilinx
Arquitectura	Multicore(DSP+ARM)	Single Core	Single Core
Costo Kit de desarrollo	<$400 dls	<$1500 dls	<$2000 dls
DMPIS del Procesador	1200	119	119
MHZ del Procesador	600	100	100
Memoria ROM (FLASH) en MB	256	16	32
Memoria RAM en MB	256	32	32

Tabla 2. Tarjetas de desarrollo

El BeagleBoard supera hasta por 10 veces en velocidad a los sistemas FPGA de Xilinx y Altera, además cuenta con un DSP de alto desempeño para procesamiento de multimedia en modo multitarea, es hasta 5 veces más económico que el Virtex-6 de Xilinx y posee soporte de gran variedad de Sistemas operativos basados en Linux. Actualmente no existe algún soft core de un procesador para FPGA con el performance del OMAP, aunque ARM ya está comenzando a introducir al mercado su tecnología para FPGA's de procesadores Cortex-M. Sin embargo, sabemos que la versatilidad que se adquiere en el diseño del

Hardware con un FPGA puede mejorar grandemente el funcionamiento de aplicaciones de uso específico como sería el caso de la implementación del protocolo J1939. Además de evitar las limitantes del MCP2515 que por contar con buffers de recepción tan pequeños y utilizar el bus de control SPI pudiera en algunos casos perder mensajes.

2.2 Estado del arte del software:

Algunas compañías como Vector y port.de ofrecen librerías que implementan el protocolo J1939, son de fácil integración y ahorran tiempo de desarrollo, sin embargo, la adquisición de estas licencias su soporte y entrenamiento eleva mucho el costo de ingeniería.

Vector :

Es una compañía que ofrece productos y servicios relacionados con diversos protocolos de comunicación y redes para aviónica, automotriz e industria en general. Entre sus productos ofrece (Hardware y Software) para los protocolos de comunicación: CAN, LIN(35), MOST(39), CANOpen, AUTOSAR(3), así como también para los estándares J1939, J1587, ISOBUS, herramientas de diseño, simulación y diagnóstico.

Vector vende una librería de J1939 desarrollada en ANSI-C (2) y adaptable a diversas plataformas de Hardware y Software como Linux embebido e incluso para el controlador de CAN MCP2515 [17]. Además provee otros productos para facilitar el análisis y desarrollo de un sistema J1939:

- CANoe.J1939: Simulación, pruebas y desarrollo..
- CANalyzer.J1939: Análisis, tráfico y adquisición de datos.

Vector es una compañía establecida en Polonia desde 1988 con amplia experiencia en el desarrollo de redes automotrices, el adquirir su aplicación para el desarrollo de esta tesis puede ahorrar mucho tiempo de desarrollo, así como proveer una solución robusta con un soporte continuo. Sin embargo, esto incluye costos de licencias de uso, así como tiempo de integración y adaptación de sus aplicaciones dentro de nuestro hardware.

Port.de:

Esta compañía, fundada en Alemania en 1990, ofrece soluciones para el mercado de embebidos que requieren la utilización de CAN, CANopen, DeviceNet (15) y Ethernet(22). También vende el Stack de J1939 desarrollado en ANSI-C como una completa implementación de SAE. Provee todos los mecanismos de comunicación definidos por la especificación permitiendo enfocar al usuario únicamente en el desarrollo de la aplicación. Un servidor de J1939 para tener acceso a la red, generar diagnósticos y pruebas. Provee soporte para diversos equipos como lo es el MCP2515 y utiliza como base de desarrollo la arquitectura del driver can4linux(10) , aunque también brinda soporte a diversos sistemas operativos [18]. Es una compañía con más experiencia en sistemas Linux que Vector por lo que podría considerarse una opción más viable.

Otros desarrollos:

A continuación se detallan cuatro desarrollos recientes que implementan el bus de CAN en un sistema de Linux embebido, el primero utiliza un procesador PowerPC MPC5200B, el segundo un procesador ARM9-S3C2410 con el controlador MCP2515, el tercero utiliza una tarjeta de desarrollo Overo (similar a la tarjeta Beagleboard) con el mismo procesador y controlador de CAN para comunicarse con dos redes. El cuarto desarrollo es la implementación del driver del controlador MCP2515 en Linux. Aunque todos presentan muy buen rendimiento y funcionalidad similar ninguno implementa el Stack de J1939.

En el documento *Design and implementation of multi-channel CAN Communication Interface based on embedded Linux [7]* se presenta una solución de un sistema Linux que utiliza un procesador PowerPC MPC5200B. En este se demuestra la factibilidad y el alto rendimiento de un driver multicanal de CAN desarrollado en Linux con el controlador de CAN Intel 82527. El sistema utiliza un FPGA Cyclone II para comunicar el procesador con los dos controladores de Red.

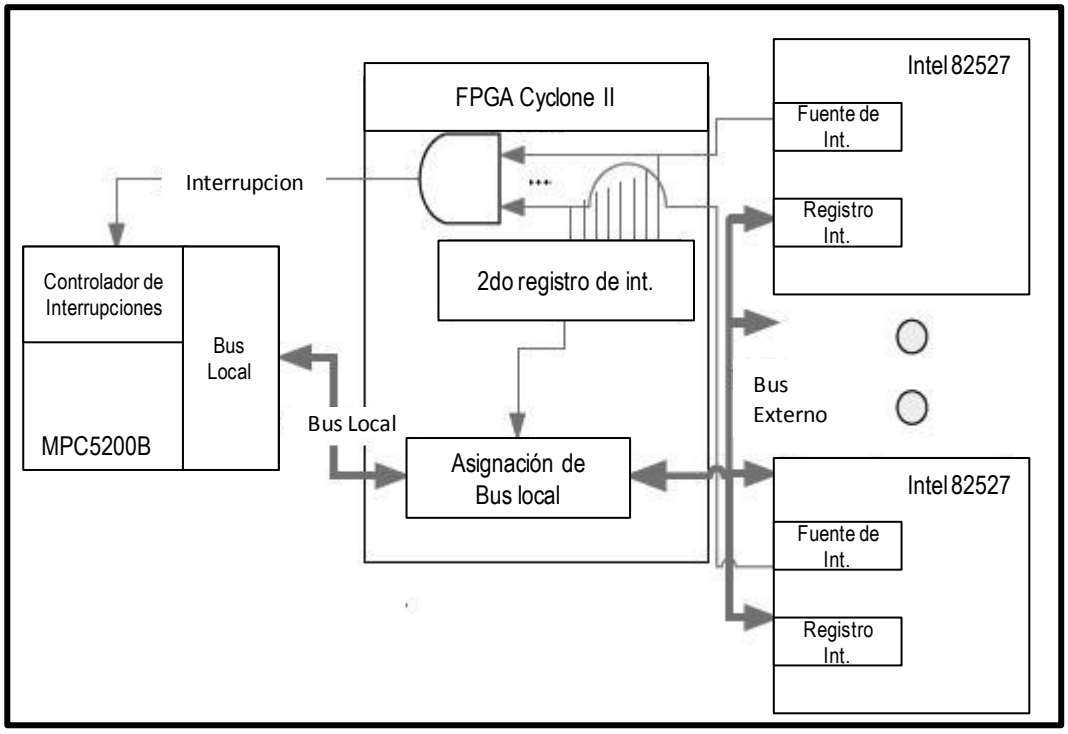


Figura 9. Conexión del bus de CAN con el procesador MPC5200B [7].

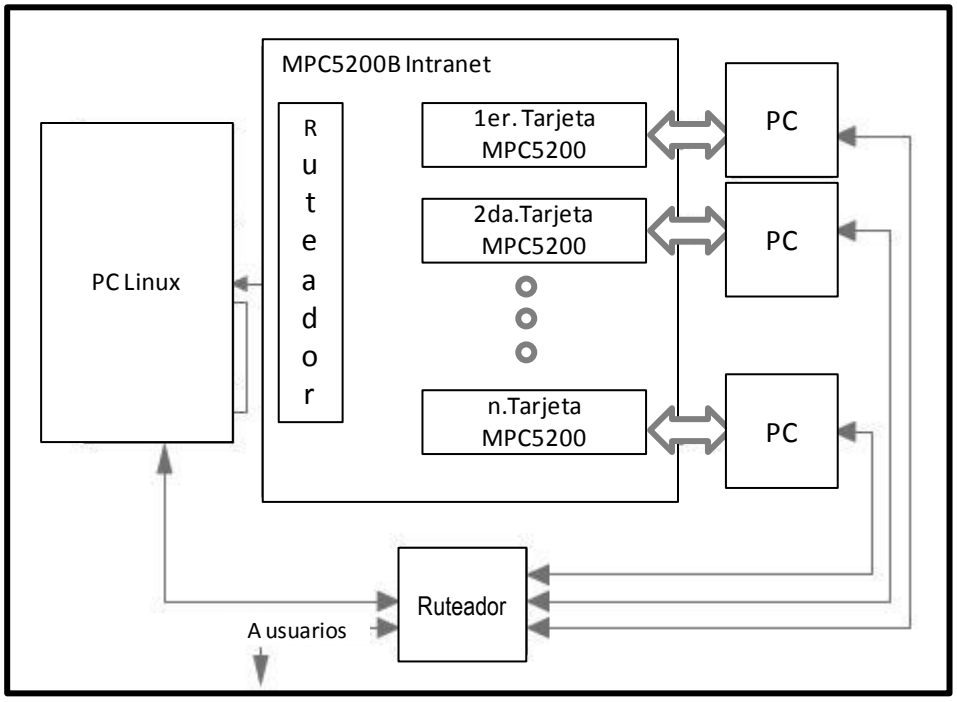


Figura 10. Arquitectura de la conexión Ethernet de cada procesador MPC5200B [7].

Cada procesador se conecta a un Ruteador para comunicarse con el servidor Linux. Utilizando el puerto serial para diagnóstico de cada procesador es posible conectarse a una PC cliente.

En el artículo *Study of LAN embebido NC system based on ARM and DSP [30]* se presenta una aplicación Linux utilizando un Arm y el controlador de CAN MCP2515. El desarrollo del Software es de igual manera en Linux y presenta grandes optimizaciones y eficiencia para el control en tiempo real de un servo motor debido a las capacidades del sistema operativo. Este sistema embebido con implementación de Red es de arquitectura simple, tiempo real, bajo costo, alta velocidad, alta eficiencia y buena conectividad.

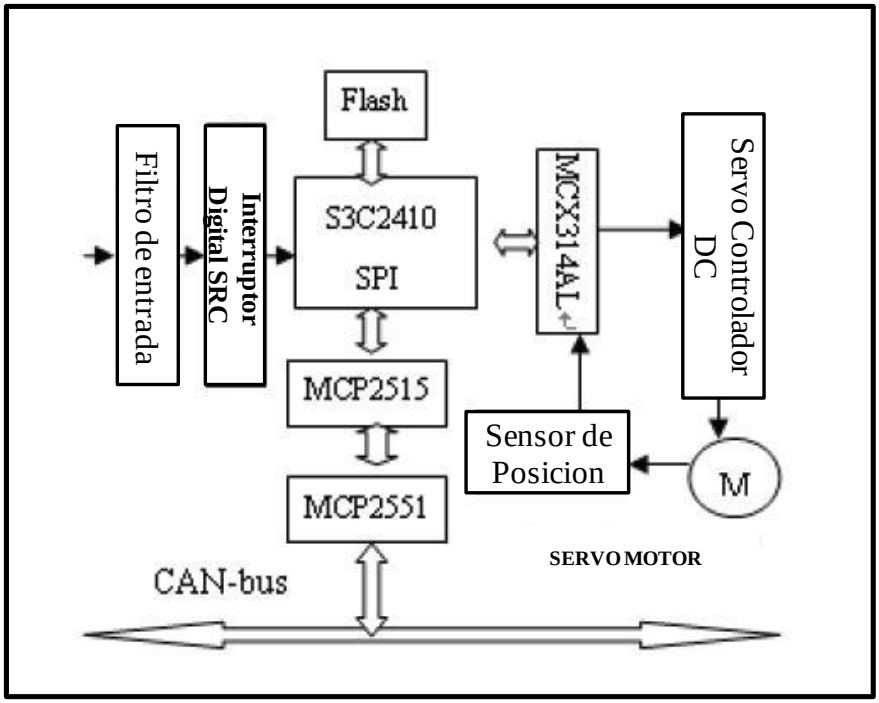


Figura 11. Sistema de Control de servo motor utilizando CAN [30].

Diseño sistema can con tarjeta de desarrollo Overo-Gumstix:

En un estudio de Tesis presentado por la *University of New South Wales* de Andrew Wrigley [3] , se detalla la utilización de una tarjeta de desarrollo (Overo) con Linux Embebido que utiliza el mismo procesador propuesto como solución de la presente investigación. También utiliza un MCP2515 para comunicarse con dos redes de control de CAN y mantener un flujo de información entre los módulos de Motor y telemetría(60).

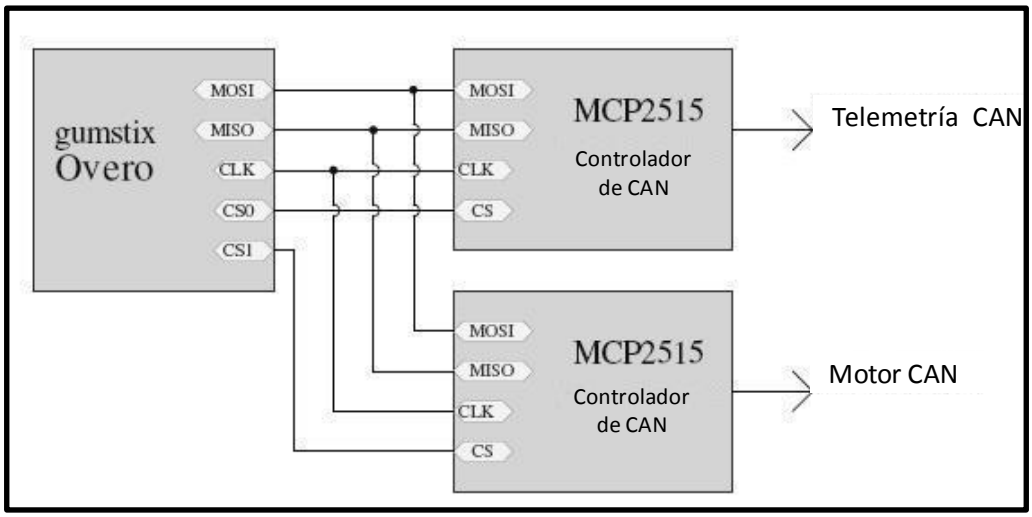


Figura 12. Conexión de dos redes de CAN a tarjeta Overo (Gumstix) [3].

Sin embargo dicha configuración presenta algunas desventajas de acuerdo al análisis de la misma tesis. Dado que la tarjeta de desarrollo Overo presenta solo 1 bus de SPI es probable perder mensajes al querer conectar 2 controladores de CAN al mismo puerto. Las limitantes del ancho de banda del puerto SPI pueden poner en riesgo el control mismo del vehículo. Dada esta situación se propone utilizar un controlador intermedio para los buses de CAN y la tarjeta de desarrollo como se muestra a continuación. Este procesador involucra mayor desarrollo de Software para realizar rutinas de control de alta prioridad que no pueden ser ejecutadas en el procesador Overo.

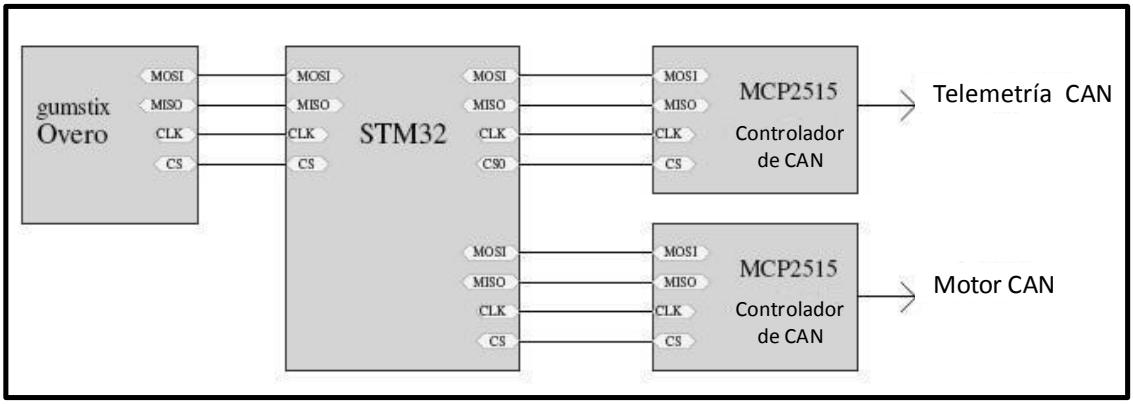


Figura 13. Conexión de dos redes de CAN a tarjeta Overo utilizando un microcontrolador intermedio STM32 [3].

Esta implementación puede ser desarrollada en la tarjeta de desarrollo Beagleboard sin requerir el procesador extra debido a que presenta dos puertos SPI independientes. Mediante este estudio podemos notar que no existe riesgo de implementar el Stack de J1939 utilizando un solo controlador de CAN MCP2515 y conectarlo directamente a uno de los puertos de SPI del procesador OMAP.

MCP2515 driver Linux open source:

Existe un desarrollo de Software Abierto para controlar el MCP2515 (mcp251x.c) [12]; sin embargo, dicho driver no presenta la posibilidad de configurar las máscaras de hardware para filtrar los mensajes de CAN y poder así implementar un Stack de J1939 eficiente.

El driver implementado procesa todos los mensajes que se reciben en el bus de CAN aunque no vayan destinados al módulo en cuestión, por lo que para implementar J1939 se tendrían que activar mascaras o filtros de Software que disminuyen grandemente la eficiencia y se corre el riesgo de perder mensajes.

Conclusiones del estado del arte del Software:

Existen soluciones comerciales confiables para implementar el Stack de J1939 que incluyen un costo y que aun así requieren esfuerzo para integrarse a cualquier arquitectura especifica. También se han presentado estudios en donde se utilizan configuraciones de Hardware y Software para integrar CAN a un sistema embebido Linux. Estas propuestas validan la factibilidad de generar una solución confiable, de bajo costo y eficiente en el mercado actual.

Por la presente investigación podemos concluir que actualmente no existe una solución de Software abierto (y/o gratuita) para implementar J1939. Este trabajo propone implementar un Building Block o librería de J1939 que sea fácilmente reutilizable, mantenible, que ahorre costos, se adapte fácilmente a un sistema embebido Linux y además brinde al usuario la capacidad de configurar filtros y/o mascaras por Hardware para obtener buena eficiencia y confiabilidad.

A continuación se presenta una tabla que resume los desarrollos previos a esta investigación:

Desarrollo, producto o estudio.	**Fecha**	**Autor**	**Descripción**
Libreria j1939 de Vector	Desde 1988	Compañía	Vende una librería j1939 en ANSI C así como utilerías y herramientas de desarrollo.
Libreria j1939 Linux de Port.de	Desde 1990	Compañía	Vende una librería j1939 para Linux, utilerías y herramientas de desarrollo.
Articulo: *"Study of LAN Embedded NC system based on ARM and DSP"*	2009	Yu Jianfeng, Jian g Tingbiao	Desarrolla un sistema de control utilizando un ARM, Linux y el controlador MCP2515 para implementar CAN.
Articulo: *"Design and implementación of multi-channel CAN Communication Interface based on Embedded Linux"*	2009	Gang Dai, Guanghua Gong, Beibei Shao, Wei Su	Desarrollo de una red de CAN multicanal utilizando Linux y Procesadores Power PC
Articulo: *"Steering-Integrated Driver Controls for Sunswift IV"*	2009	Andrew Wrigley	Desarrolla un sistema embebido Linux para control de vehículo eléctrico implementando CAN con el controlador MCP2515 en un ARM.
Driver abierto para Linux: MCP251X.c	2009	Chris Elston	Implementa un Driver abierto para implementar CAN utilizando el controlador MCP2515 para sistemas embebidos Linux.
Librería J1939 para Linux	2011	Luis Arturo Lopez	Implementación de librería j1939 para Linux utilizando el Controlador de CAN MCP2515

Tabla 3. Desarrollos de CAN y J1939 en sistemas Linux

3. METODOLOGIA

Para poder implementar este producto es necesario seguir un proceso de desarrollo que incluye varias etapas que van desde la integración del Hardware hasta el desarrollo de la aplicación que prueba toda la funcionalidad de la librería.

Hardware:

-Tarjeta de desarrollo: Seleccionar tarjeta que soporte Linux embebido con Microcontrolador ARM de preferencia y que soporte al menos un puerto de SPI.

-Tarjeta SPI-CAN: implementar tarjeta con los controladores de CAN MCP2515/10 y MCP2551.

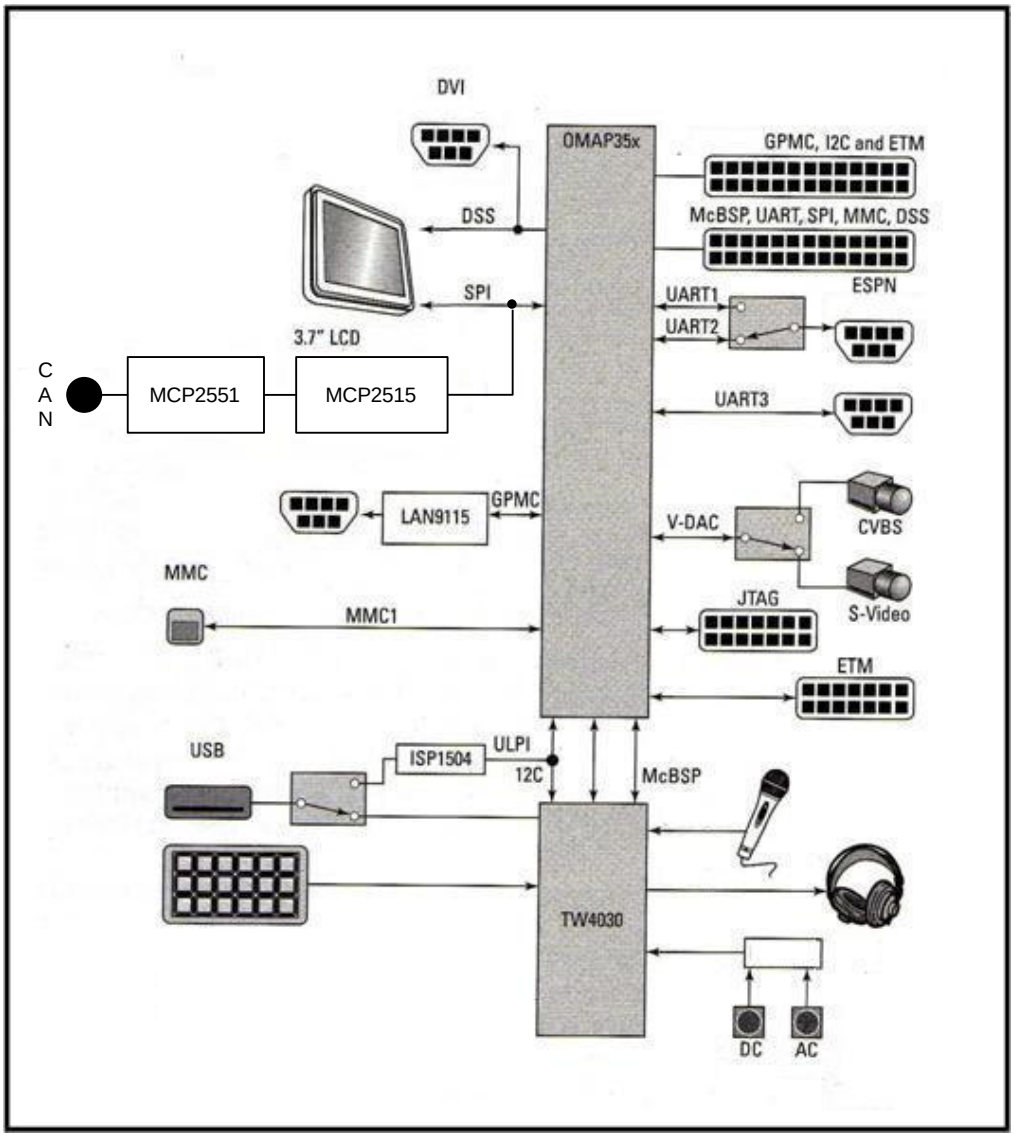


Figura 14. Implementación de CAN para la tarjeta de desarrollo OMAP3 EVM [22].

Software:

-Implementación de drivers de SPI y CAN para controlar MCP2515/10.

-Implementación del Stack SAEJ1939. (Producto principal).

-Aplicación: Con el fin de probar el sistema se generara una simulación que recibirá mensajes de un controlador (SWC) y mandara mensajes de estatus a un monitor de CAN.

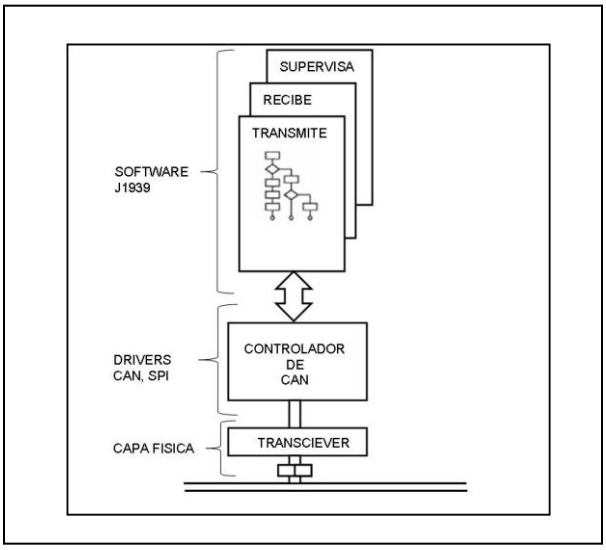


Figura 15. Implementación de Software.

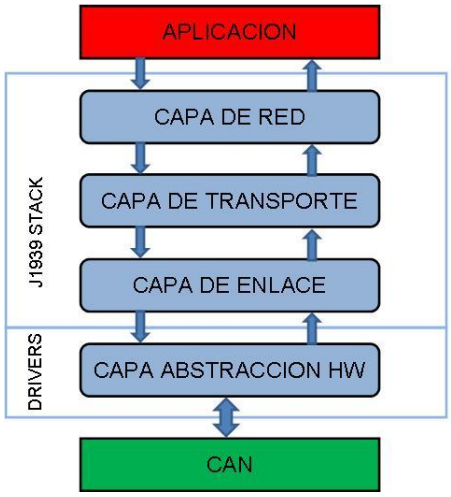


Figura16. Capas OSI(44) del Stack J1939 [18].

El proceso de desarrollo del sistema puede ser dividido en 5 etapas por lo que se seguirá una metodología de desarrollo incremental en donde cada etapa entregable pueda ser verificable [20] y con esto poder cubrir las capas que abstrae J1939. En la primer etapa se integrara el Hardware, que implica la fabricación y armado de la tarjeta interfaz de CAN, así como su interfaz con la tarjeta de desarrollo de Linux embebido (ver Figura.14). Para verificar el funcionamiento de los buses en la tarjeta se utilizara monitores de SPI y CAN instalados en una PC. En la segunda etapa se implementara la capa de abstracción de Hardware (ver Figura 15) en donde se integraran los drivers de SPI y CAN para Linux. Se programaran aplicaciones de prueba que midan el rendimiento y verifiquen la funcionalidad SPI y CAN respectivamente. La tercera etapa abstrae la capa de enlace de J1939. Una aplicación podrá configurar J1939 para transmitir y recibir mensajes midiendo su rendimiento de acuerdo a la especificación. La cuarta etapa incluye la capa de transporte de J1939 y de la misma manera que la tercera etapa es necesario crear una aplicación que permita enviar y recibir mensajes utilizando la capa de transporte. En la Quinta etapa se incluye la capa de aplicación que

configura y utiliza J1939 para comunicarse con otros periféricos. Para validar el sistema completo la aplicación deberá de ser capaz de comunicarse con 2 módulos de CAN, como se mencionó anteriormente un SWC y un monitor de CAN. Se diseñaran métodos de prueba que verifiquen la funcionalidad del sistema.

Para obtener una solución práctica y reutilizable en sistemas PC de escritorio con el mismo propósito se propone utilizar el puerto UART de la PC como puerto de comunicación de CAN. Para esto es necesario utilizar un procesador PIC intermedio que convierta los comandos de UART a SPI y viceversa. La solución propuesta se muestra a continuación:

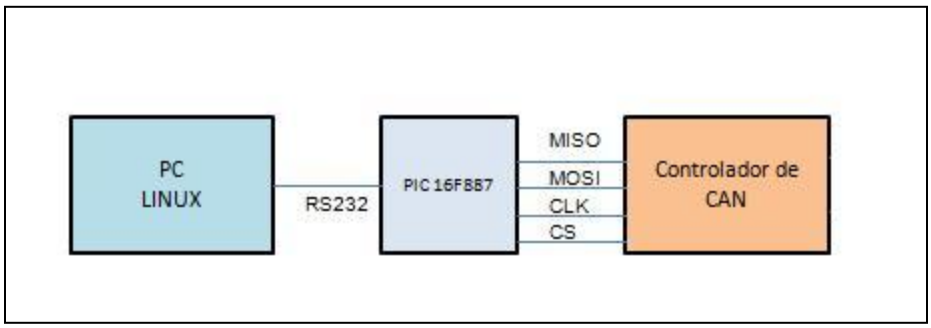


Figura 17. Conexión de una PC al controlador de CAN MCP2515

Una de las ventajas de este desarrollo es que se puede desarrollar el software de una manera más simple, compilando y probando en la PC disminuyendo grandemente el tiempo de desarrollo del mismo. El diagrama de flujo muestra la implementación de la aplicación específica requerida para el PIC 16887 [14].

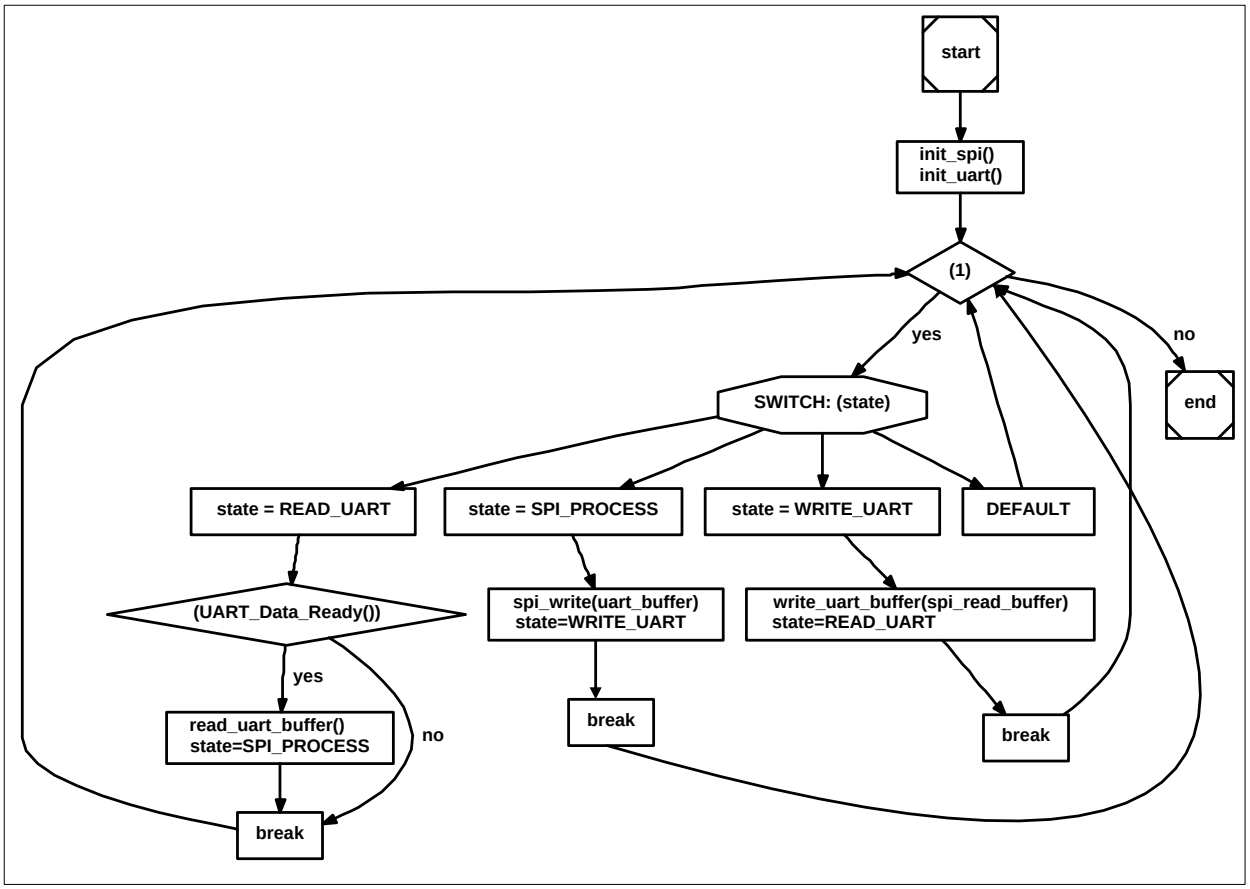


Figura 18. Diagrama de flujo para convertir SPI-UART utilizando el PIC16887

3.1 Etapas de desarrollo del sistema:

3.1.1 Etapa 1. Hardware:

Los componentes del Hardware incluyen la tarjeta de desarrollo Beagleboard, así como sus periféricos: ratón, teclado, monitor, tarjeta SD, HUB(29) de USB, regulador de voltaje.

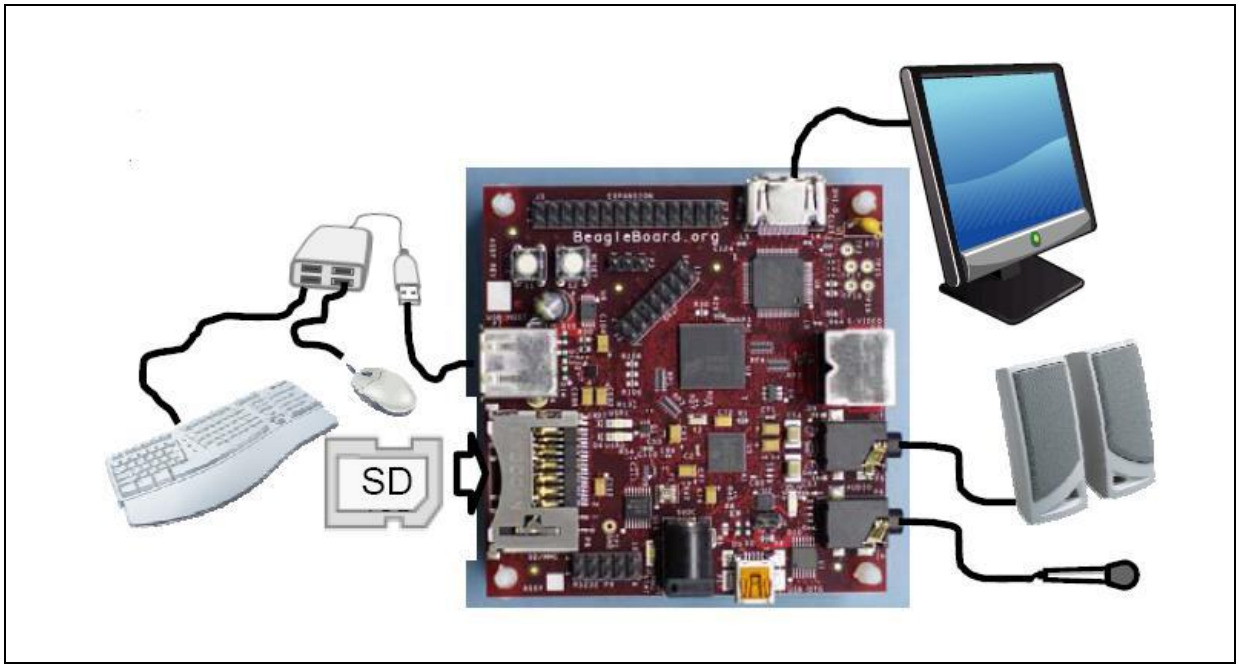


Figura 19. Conexión de Tarjeta Beagleboard con sus periféricos [4].

Para la implementación de CAN se utilizara a tarjeta de de desarrollo con el controlador de CAN MCP2515 y el Transciever(62) MCP2551.

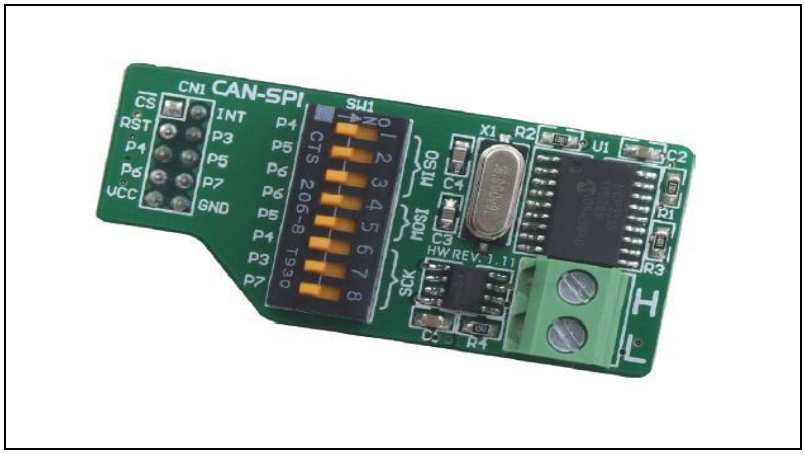


Figura 20. Tarjeta de desarrollo CAN_SPI [15].

a) Requisitos:

Alimentación de tarjeta Beagleboard 5V

Alimentación de tarjeta MCP2515 5V

Nivel de Salida de Voltajes de Beagleboard SPI 1.8V

Nivel de entrada de Voltajes MCP2515 5V

La Velocidad del reloj de SPI puede ser configurada de 1Mhz.

La configuración de los Modos SPI puede operar como 0,0 o 1,1:

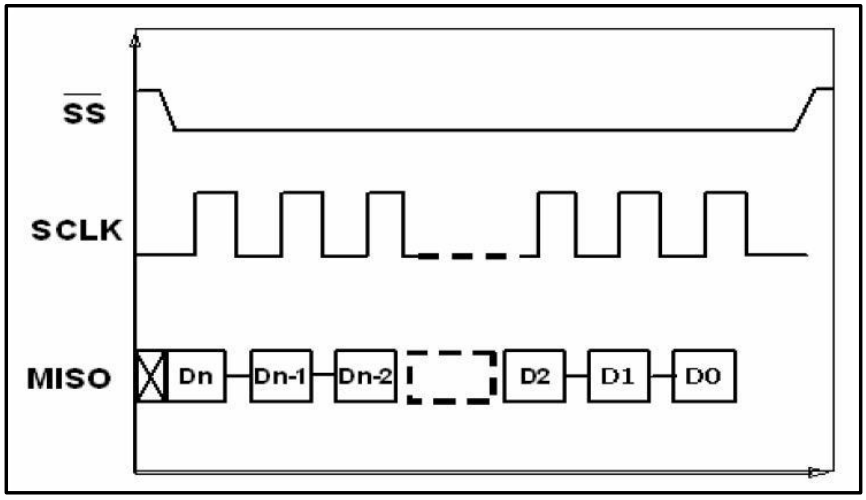


Figura 21. Descripción del Modo 0,0 SPI [6].

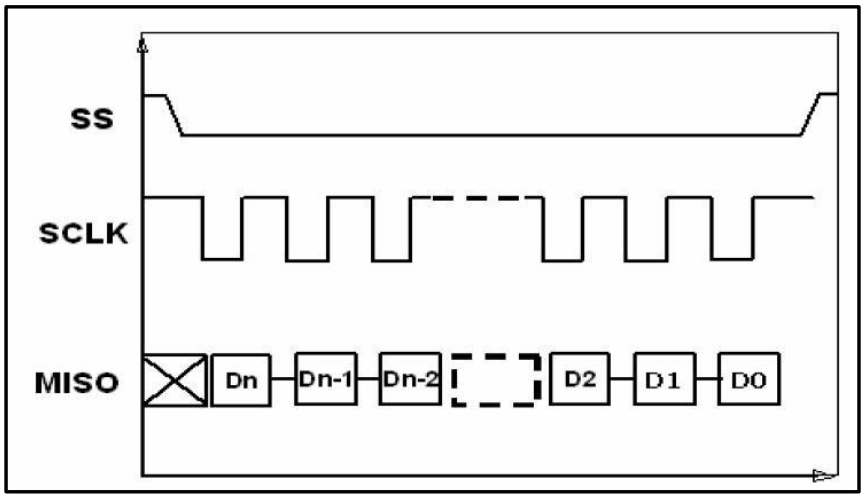


Figura 22. Descripción del Modo 1,1 SPI [6].

b) Diseño:

Para tener una conexión apropiada entre ambas tarjetas es necesario convertir las salidas del puerto SPI4 de 1.8V a 5V y así interconectarse con el nivel de voltaje del controlador MCP2515. Para esto se utilizara el convertidor de Voltaje TXS0108EPWR. A continuación se presentan los esquemáticos que explican esta conexión así como el diagrama de bloques que explica dicho diseño.

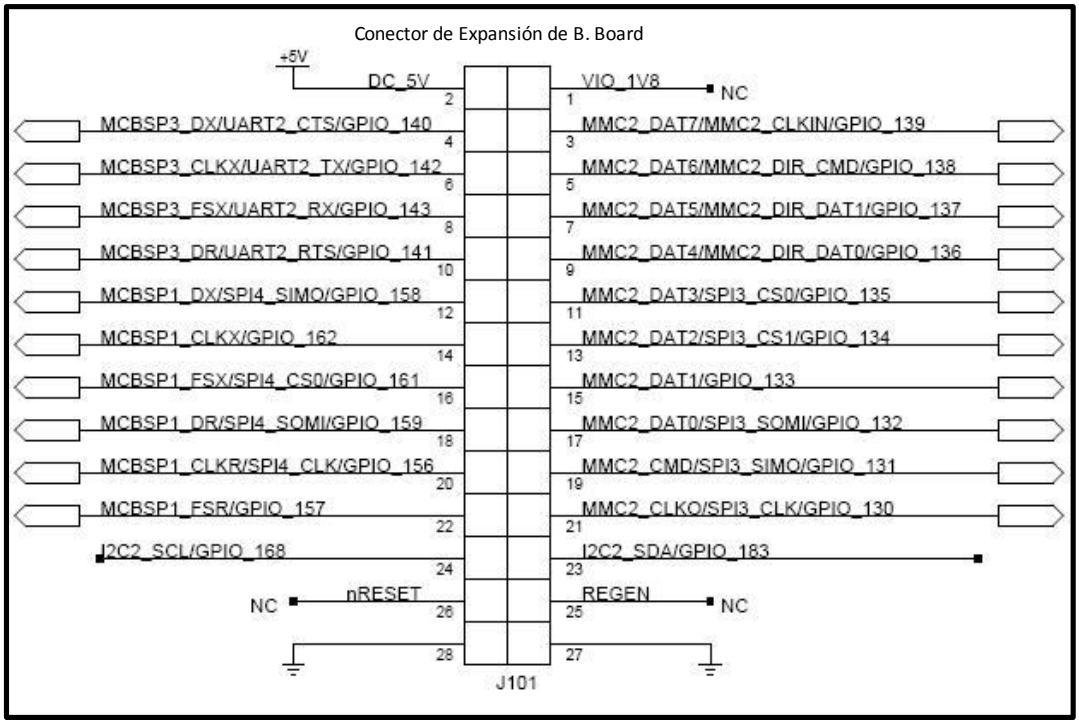


Figura 23. Puerto de Entradas/Salidas la Tarjeta Beagleboard [4] .

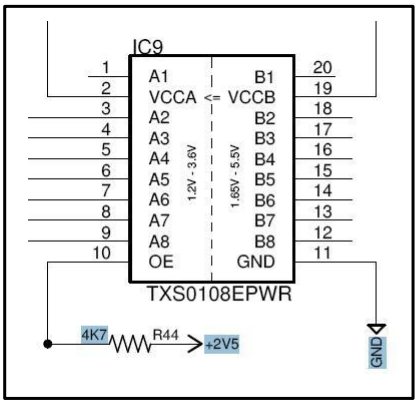


Figura 24. Convertidor de Voltaje TXS0108EPWR [24].

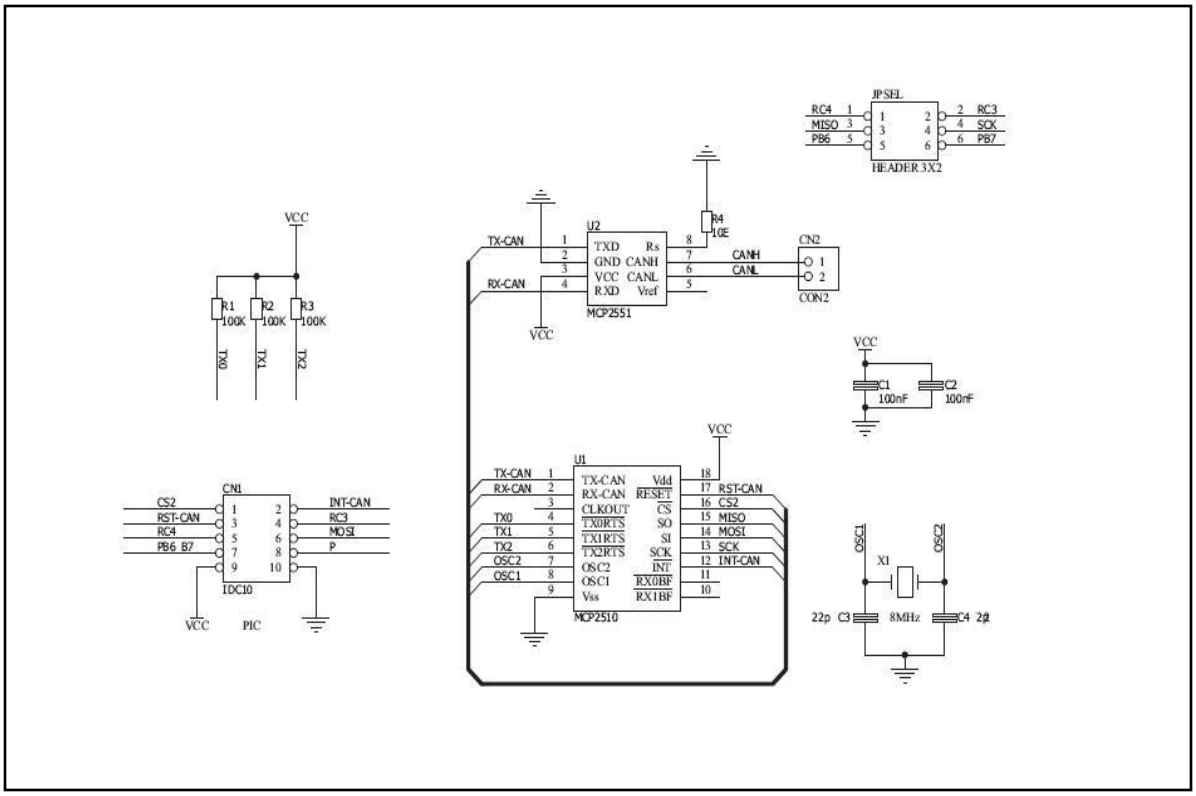


Figura 25. Tarjeta de desarrollo MCP2515 [15].

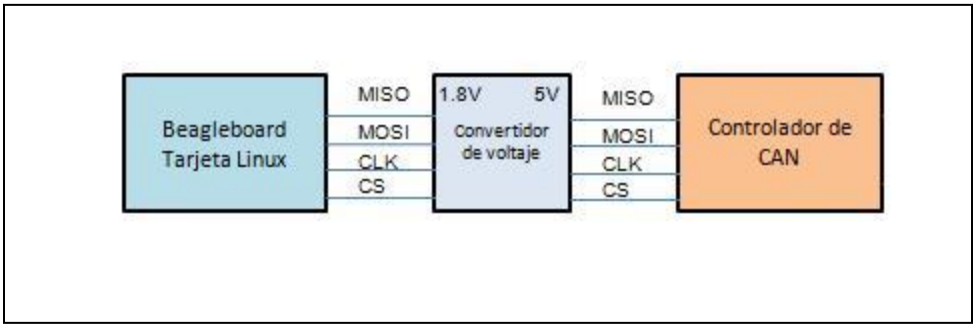


Figura 26. Diagrama de bloques eléctrico del sistema.

c) Implementación:

Desarrollo driver de SPI en Linux:

Existen 2 opciones para utilizar el SPI en un sistema embebido. Una es desarrollar un driver especifico que corra en el Kernel, o bien utilizar un driver genérico como el spidev que permite accesar al puerto SPI desde el espacio de usuario de Linux [16]. Por cuestiones de simplificar y acelerar el desarrollo se procede a utilizar el driver desarrollado para Linux embebido spidev. Existen gran cantidad y variedad de tutoriales para habilitar el puerto SPI en la tarjeta Beagleboard, uno de los más completos y comprensibles es el titulado *Writing a Linux SPI driver [28]*.

Para activar dicho driver en el sistema Linux con la tarjeta de desarrollo es necesario seguir los siguientes pasos:

1) Activar el driver en el Kernel: Para activar el spidev driver es necesario cambiar la configuración del Kernel abriendo el archivo defconfig (presente en el código fuente para compilar el Kernel). Y confirmando que se encuentran habilitadas las siguientes opciones:

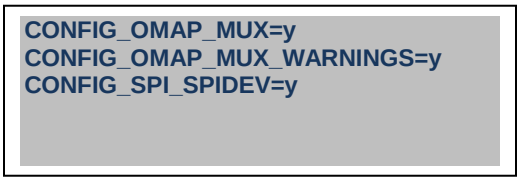


Figura 27. Configuracion de SPIDEV en defconfig file.

Después de habilitar dichas opciones será necesario guardar el archivo antes de recompilar el Kernel.

2) Definir los puertos de SPI del OMAP: Los puertos de SPI 3 y 4 se configuran mediante el MUX (multiplexor de Entradas y salidas)(42) en el archivo board_omap3beagle.c. Primeramente se definen las características generales del bus de datos (Anexo B.1) y posteriormente los puertos de entrada y salida (Anexo B.2). Para hacer estas actualizaciones al archivo es necesario generar un parche (patch file)

que modifique el archivo fuente al momento de compilar el Kernel. Este Parche debe de ser agregado a la lista de parches del sistema de archivos en: linux-omap-psp_2.6.32. El parche se encuentra definido en el (Anexo B.3). Para hacer efectivo este parche en el sistema de archivos se modifica el archivo linux-omap-psp_2.6.32 (Anexo B.4). Después es necesario guardar el archivo 0009-spidev-patch.patch en la carpeta //beagleboard-xmc/0009-spidev-patch.patch.

3) Recompilar el Kernel: Después de realizar los pasos anteriores es necesario recompilar el Kernel. Los pasos para obtener el código fuente, compilar o recompilar el Kernel fueron tomados de la página de soporte de Angstrom [9].
   El sistema operativo Angstrom posee una configuración básica y optimizada de un sistema operativo Linux con ambiente grafico para gran variedad de aplicaciones y con configuración específica para la tarjeta de desarrollo Beagleboard.

   Angstrom fue comenzado por un pequeño grupo de personas que trabajaron para OpenEmbedded, OpenZaurus y OpenSimpad, proyectos que unifican su esfuerzo para hacer una estable y amigable distribución para sistemas embebidos como handhelds(26), celulares, cajas de control etc. Angstrom es versátil y puede ser instalado en equipos con hasta 4MB de memoria flash. Esta distribución no tiene licencia, solo se necesitan evaluar que módulos y paquetes se requieren utilizar.

   Una vez recompilado el código es necesario configurar, particionar y cargar las imágenes a la unidad SD, para esto se utilizó el procedimiento documentado en el manual: "*BeagleBoard System Reference Manual" [4].*

4) Detectar el puerto SPI en el espacio de usuario. Para determinar si el driver spidev se encuentra habilitado una vez arrancado el sistema operativo, los archivos spidev3 y spidev4 deberán estar presentes en el folder raíz de /dev/ (devices).

5) Generar aplicación de prueba para spidev. La aplicación de prueba spidev_test.c de MontaVista Software configura un puerto SPI envía y despliega los bytes procesados del puerto.(Ver Anexo B.5).

   Para compilarlo se puede utilizar un Crosscompiler(13) de GCC para Arm o mediante la aplicación de GCC presente en el sistema de archivos del Kernel específico para la tarjeta de desarrollo.

   El archivo spidev_test es copiado a la SD en /home/ para después arrancar el sistema y ejecutar el programas como ./spidev_test.

   La función transfer es llamada en la función main con la dirección del puerto SPI fd, ya inicializado previamente. En la función transfer se definen el buffer de escritura tx[] y lectura rx[], estos buffers conforman la estructura spi_ioc_transfer, A través de esta estructura se tiene acceso al puerto mediante la función ioctl(fd, SPI_IOC_MESSAGE(1), &tr) (Ver Anexo B.5).

d) Verificación:

SPI está habilitado en tarjeta Beagleboard:

Se detectó que los archivos spidev3.0 y spidev4.0 se encuentran presentes en el sistema de archivos en la dirección root/dev/.

```
root@beagleboard:/dev# ls
audio           psaux           tty34
block           ptmx            tty35
bus             pts             tty36
char            ram0            tty37
console         ram1            tty38
cpu_dma_latency ram10           tty39
disk            ram11           tty4
dsp             ram12           tty40
fb              ram13           tty41
fb0             ram14           tty42
fb1             ram15           tty43
fb2             ram2            tty44
full            ram3            tty45
i2c-1           ram4            tty46
i2c-2           ram5            tty47
i2c-3           ram6            tty48
initctl         ram7            tty49
loop2           spidev3.0       tty55
loop3           spidev4.0       tty56
```

Figura 28. Despliegue de spidev3.0 y spidev4.0 presentes en tarjeta de desarrollo.

El puerto SPI envía y recibe datos en tarjeta Beagleboard:

Se corrió la aplicación ./spidev_test y se retroalimento la salida MOSI con MISO, obteniendo que los datos escritos coinciden con los leídos.

```
root@beagleboard:/home#
[  262.117248] spidev spi4.0: DMA RX last
word empty00000 Hz (500 KHz)
FF FF FF FF FF FF
40 00 00 00 00 95
FF FF FF FF FF FF
FF FF FF FF FF FF
FF FF FF FF FF FF
DE AD BE EF BA AD
F0 0D
root@beagleboard:/home# ./spidev_test4
```

Figura 29. Datos escritos y recibidos por el puerto de SPI.

Los niveles de voltaje para conectar el MCP2515:

Se verifico que los voltajes después del convertidor se encuentren en el nivel apropiado de 5V para lograr la conectividad con el controlador de CAN:

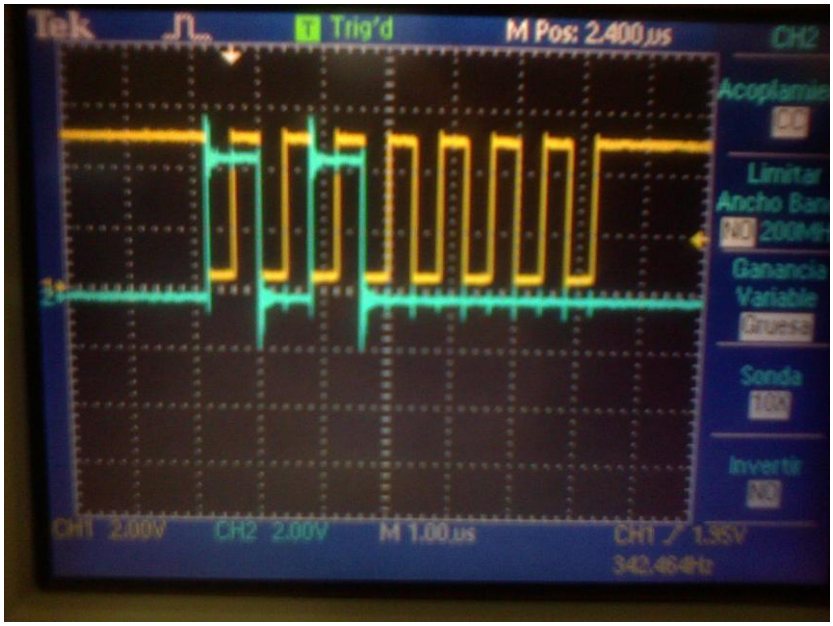


Figura 30. Señales de SPI en Tarjeta Beagleboard (@ 5V).

Se observan los pulsos de reloj en amarillo y los pulsos de datos en azul. La frecuencia del oscilador es de 1Mhz, la amplitud de las señales es de 4,7V. Lo que nos muestra que ya es posible conectarse al MCP2515.

3.1.2 ETAPA 2. Driver de CAN y SPI:

a) Requisitos

Generales de CAN:

Implementar CAN V2.0B a 250Kb/s en modo extendido

La cantidad de bytes por mensaje puede variar de 0 a 8 bytes.

-Capa Física:

La comunicación exige un cable blindado par trenzado con resistores terminadores en cada final de nodo.

Las conexiones requieren de 3 conectores definidos como CAN_H, CAN_L y blindaje a tierra.

La capa física no es tolerante a fallas.

Se permiten 30 nodos por segmento.

-Procesamiento de mensajes:

Los dispositivos deben de procesar 100% de los mensajes en el bus.

Deben de generar respuesta en menos de .20 s.

Pueden esperar respuesta hasta 1.25ms

La estructura del mensaje se define en la Figura 31.

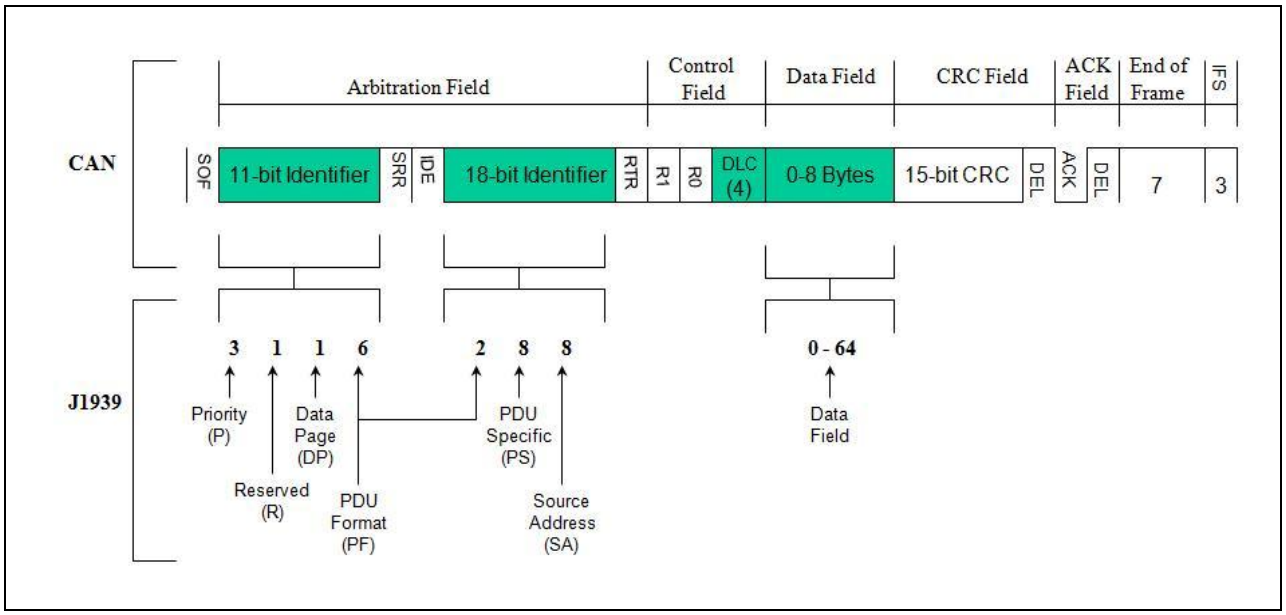


Figura 31. Formato extendido de un mensaje de CAN.

b) Diseño:

A continuación se describen mediante diagramas de estados y de secuencia UML el diseño del driver de CAN-SPI:

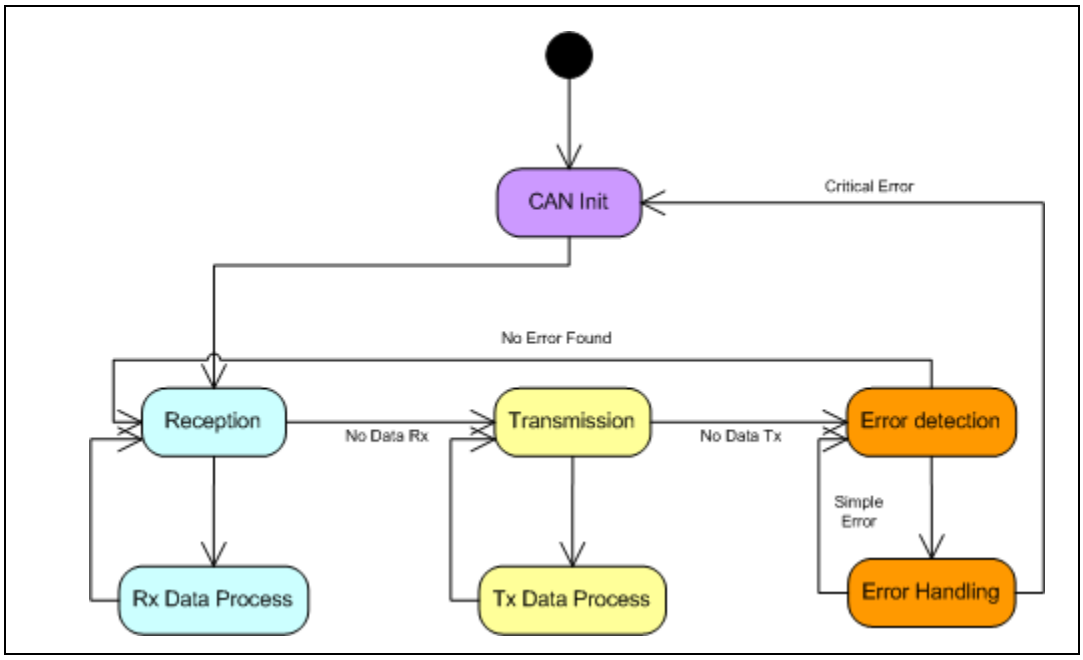


Figura 32. Diagrama de Estados UML del Driver de CAN.

Mediante el proceso de Inicialización de CAN se configura e inicializa el estado del controlador de CAN MCP2515. Una vez configurado comienza el proceso de Recepción de Mensajes en el cual el sistema pregunta si se ha recibido un mensaje, una vez recibido el mensaje es guardado en una cola de mensajes para después ser procesado por la aplicación. Posteriormente comienza el proceso Transmisión en donde se pregunta si existe un mensaje en la cola de datos de transmisión y si es posible transmitir ese dato mediante CAN. Después de transmitir el mensaje comienza el proceso de detección de errores en donde se detectan los errores de CAN que en caso de ser errores críticos el sistema se reinicializara, en caso contrario se regresa al proceso de recepción.

Diagrama de secuencia UML:

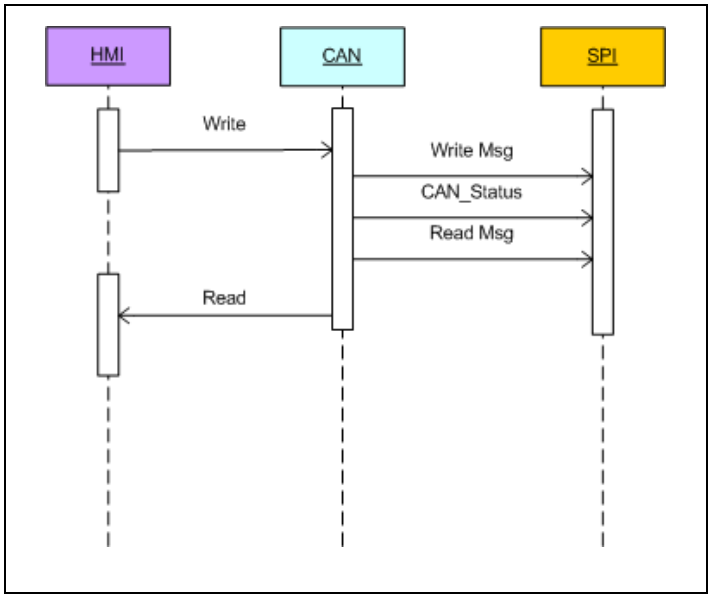


Figura 33. Diagrama de Secuencia UML del Driver de CAN-SPI.

El proceso de CAN recibe un mensaje mediante la aplicación HMI(28). Para transmitir ese mensaje mediante SPI es necesario escribir un comando de escritura y proceder al envió de bytes del mismo mensaje. Posteriormente para leer un mensaje es necesario preguntar el estatus del controlador, una vez sabiendo que se recibió un mensaje, se enviara el comando de lectura para que el mensaje sea adquirido byte por byte y después sea reenviado a la aplicación HMI.

c) Implementación:

Se muestran 2 diagramas de flujo el primero implementa la función J1939_ReceiveMessageque se encarga de verificar si el controlador de CAN ha recibido algún mensaje en alguno de los 2 buffers de recepción. En caso de haberlo recibido el mensaje es procesado leyendo los 14 bytes que lo conforman desde el puerto SPI. En el segundo diagrama de flujo se implementa la función de transmisión o envió de mensajes. En donde primeramente se verifica el estado de los buffers de escritura, si alguno se encuentra disponible se procede a transmitir los 14 bytes del mensaje por el puerto SPI.

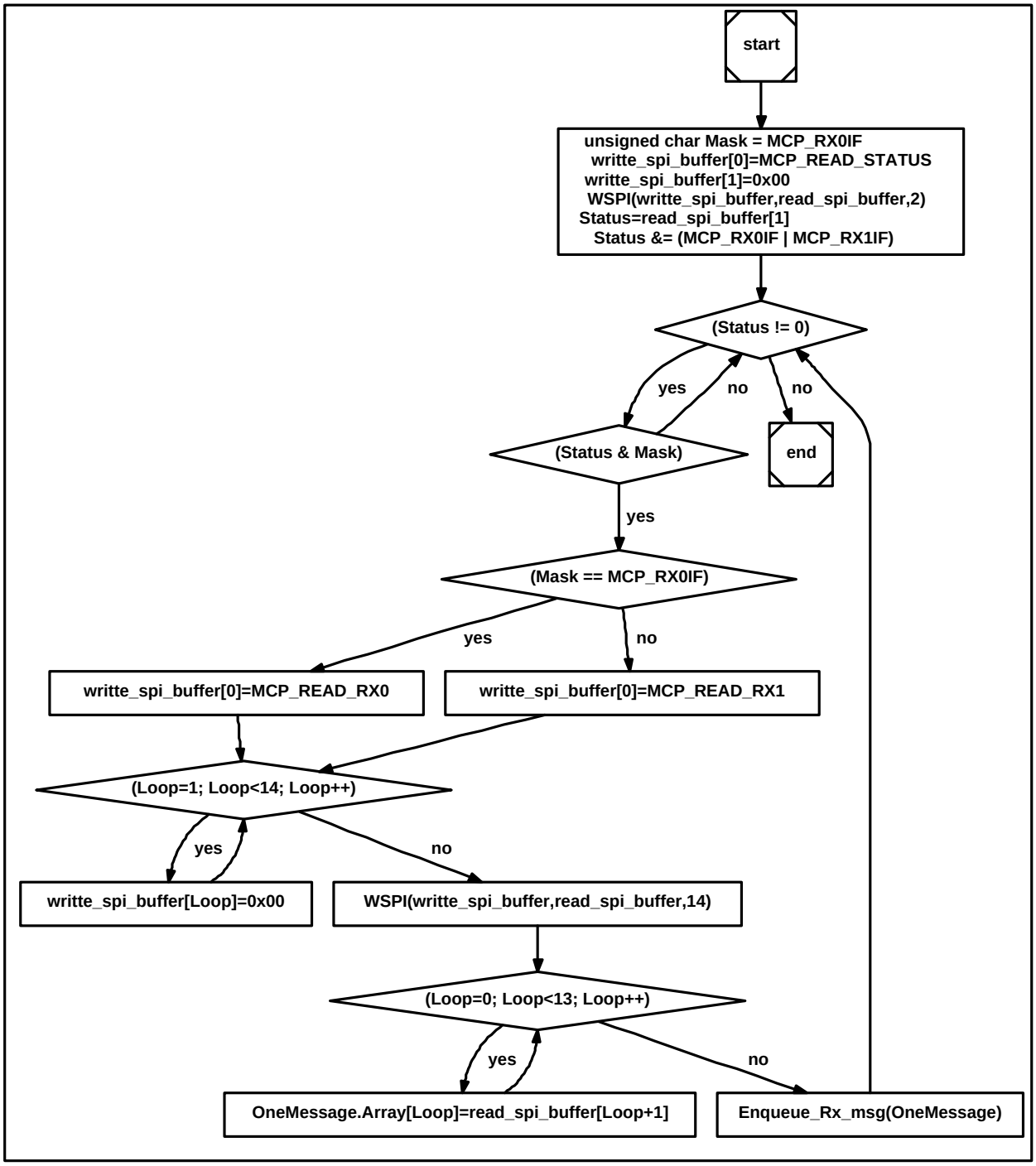


Figura 34. Implementación de la función de recepción de mensajes ReceiveMessage( ).

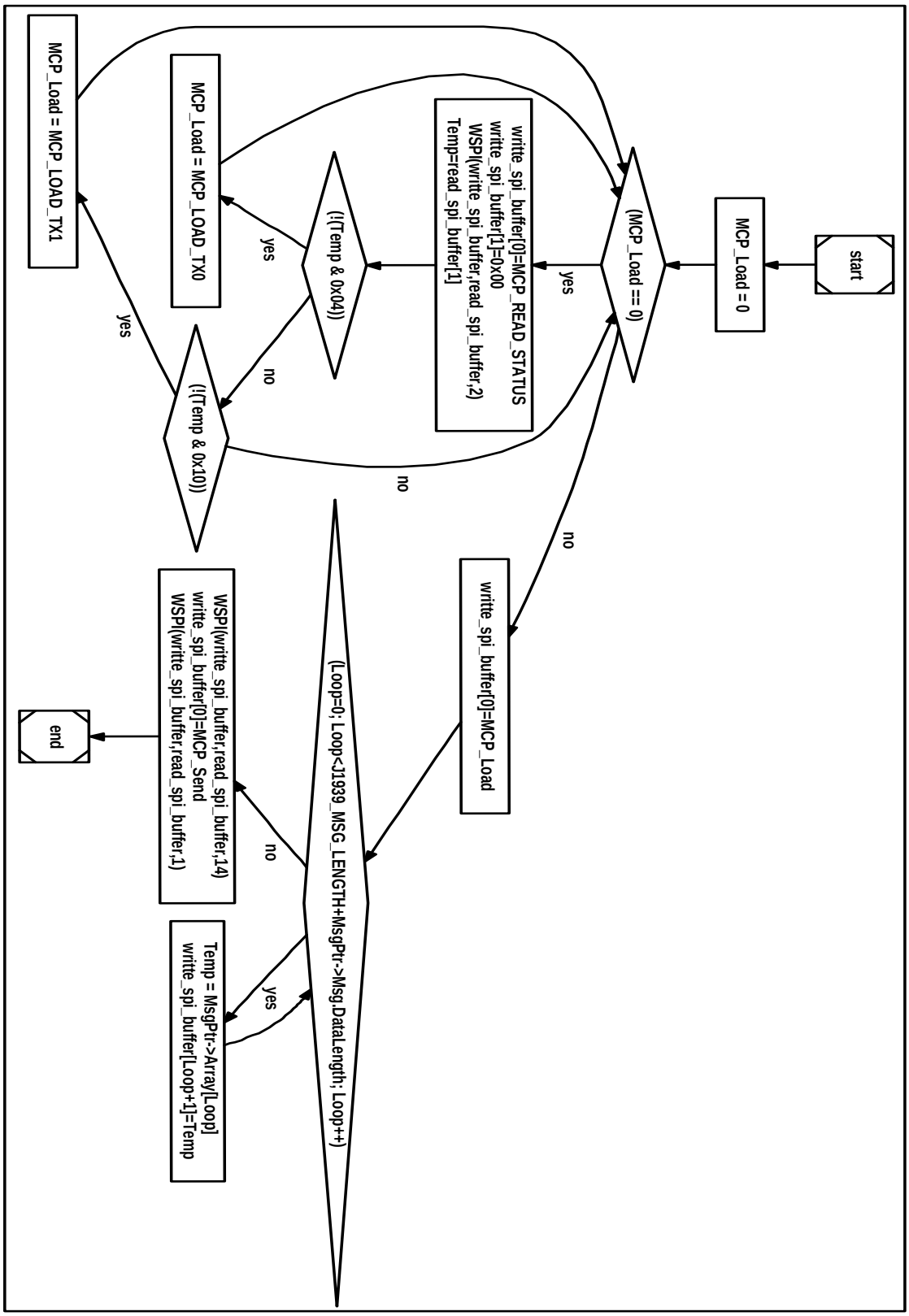


Figura 35. Implementación de la función de transmisión de mensajes SendMessage(MESSAGE *MsgPtr).

d) Verificación:

En esta etapa de desarrollo se verifica que la comunicación se encuentre habilitada entre la tarjeta de desarrollo y una herramienta para monitoreo de mensajes. Con esto se comprueba la funcionalidad de la capa física implementada en la tarjeta MCP2515.
Esta tarjeta utiliza 2 integrados para cumplir con la especificación de CAN:

MCP2515:

El controlador implementa por completo la especificación de CAN versión 2.0B. Es capaz de transmitir y recibir mensajes en formato estándar y extendido de manera simultánea. Posee 2 mascaras de recepción y 6 filtros de recepción para delimitar el tipo de mensajes a procesar y con esto disminuir el tiempo de procesamiento de mensajes no utilizados. El módulo de CAN maneja todas las funciones de transmisión y recepción de mensajes del bus. Cada mensaje detectado es verificado para que esté libre de errores de CAN antes de pasar por el filtrado definido por el usuario. Cada error es detectado y contabilizado para realizar su monitoreo y el manejo de mismos.

MCP2551:

El transciever funciona como interfaz entre el controlador de CAN y el bus físico. El MCP2551 provee la transmisión y recepción diferencial por lo que es completamente compatible con el estándar ISO-11898, incluyendo los requerimientos de 24V. Opera a velocidades de hasta 1Mb/s.

Para verificar que se reciben y transmiten mensajes se utilizó un monitor de CAN comercial conocido como Santo (Saint Box) que se conecta a la PC por medio de USB. A continuación se presentan los mensajes transmitidos y recibidos.

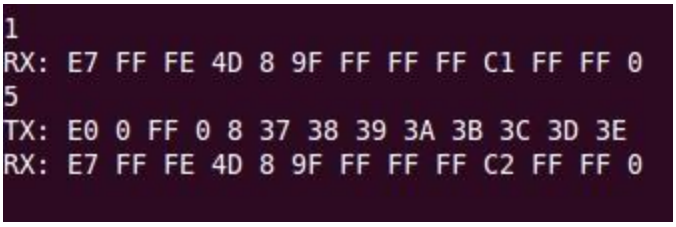


Figura 36. Mensajes recibidos y transmitidos por el driver de CAN.

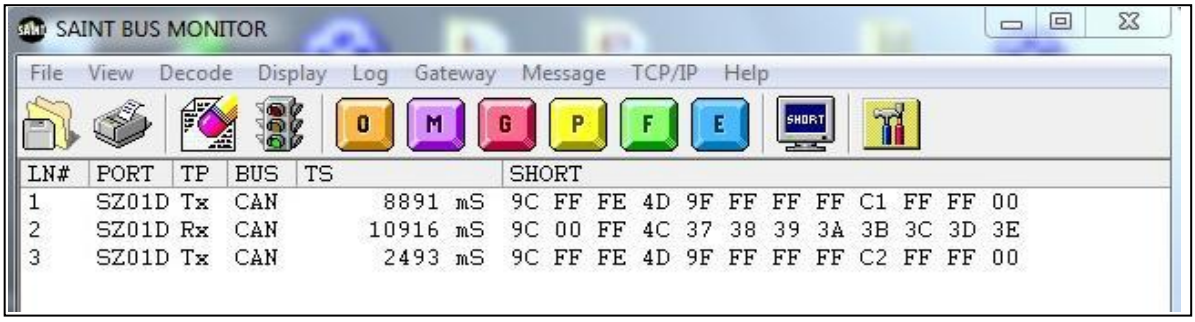


Figura 37. Mensajes recibidos y Transmitidos por el Santo (al driver de CAN).

Se observa que los datos transmitidos y recibidos son los mismos (últimos 8 bytes). El formato de los primeros 5 bytes es presentado en un orden distinto por lo que no coinciden al 100%. El objetivo de esta etapa es verificar que existe comunicación de CAN por lo que no se profundizara para explicar estas diferencias.

3.1.3 ETAPA 3. Capa de enlace de datos:

Esta capa define el uso y manejo de la estructura de los mensajes de CAN específicos del protocolo J1939.

a) Requisitos:

Estructura de un PGN (Parameter Group Number), Parámetro de Grupo que identifica el tipo y modo del mensaje de J1939:

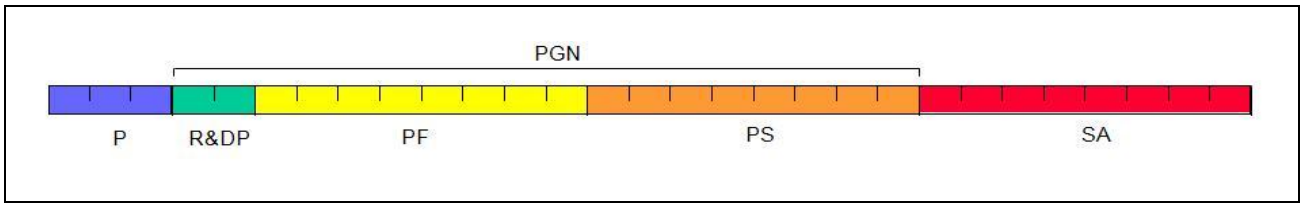


Figura 38. Estructura de un mensaje de J1939 (PGN(45)).

En donde:

(P) Prioridad del mensaje en donde 000 es la prioridad más alta y 111 es la más baja

(R) Reservado para uso futuro

(DP) Selecciona entre las 2 páginas posibles de datos

(PF) Formato: Determina el formato del PGN el tipo y modo del mensaje.

(PS) Formato especifico – (dependiente del valor PF) , puede contener la dirección destino del mensaje o un valor especifico del PGN.

(SA) Dirección del nodo que envía el mensaje

El número máximo posible de PGNS es de 8672 y su uso se encuentra predefinido por SAE.

-Mensajes de destino específico: Su dirección destino es requerida PDU1 va desde 0 hasta 239. Cuando la dirección DA es 255 el mensaje es Global y se envía a todos los nodos.

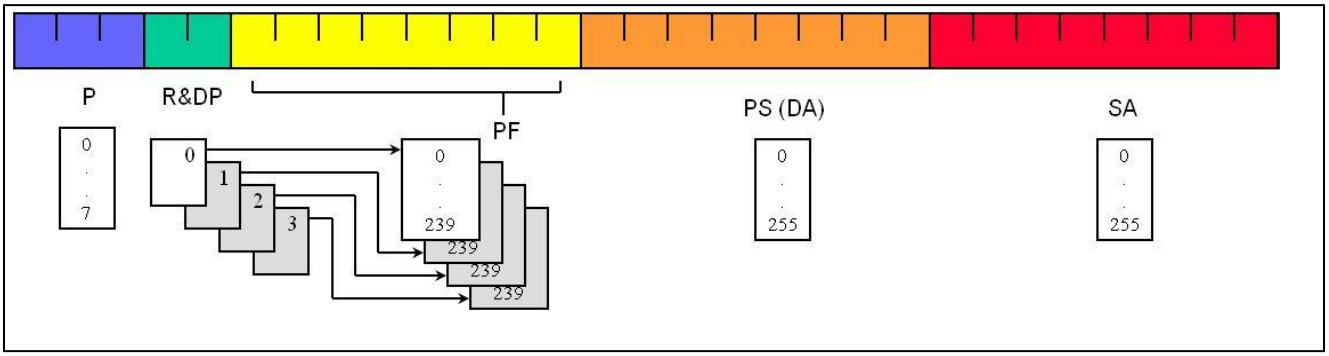


Figura 39. Mensajes de J1939 de destino específico.

-Mensajes de Grupo extendido (Broadcast) PDU2. En donde los valores de PF se definen entre 240 y 255. Se utilizan para enviar mensajes de múltiples fuentes a múltiples destinos.

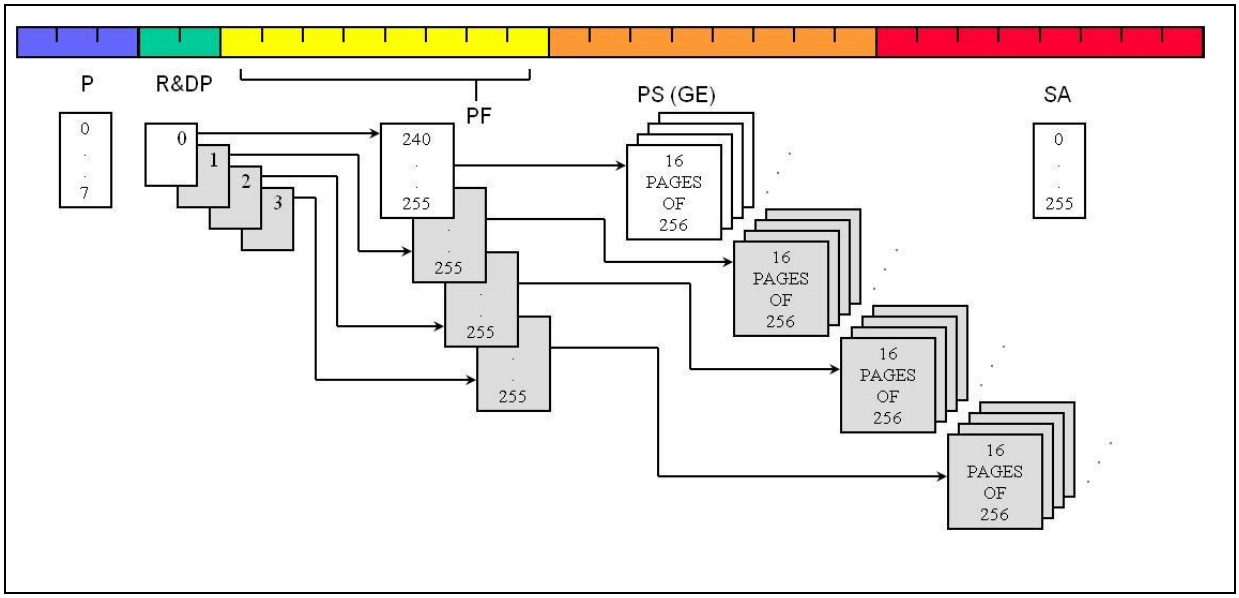


Figura 40. Mensajes de J1939 de Grupo Extendido.

b) Diseño:

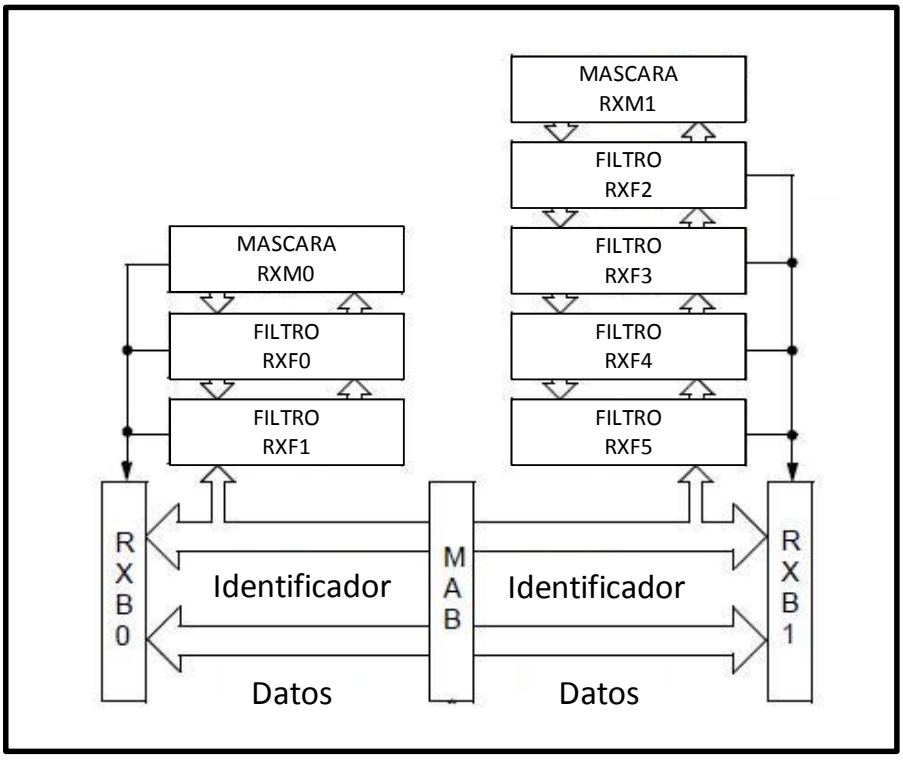


Figura 41. Recepción de los buffers(7) de CAN de controlador MCP2515 [13].

El controlador de CAN utiliza 2 buffers de recepción de mensajes. La recepción de mensajes en el Buffer RXB0 puede ser configurada utilizando 1 mascara RXM0 y 2 filtros RXF0, RXF1. El buffer RXB1 utiliza una máscara RXM1 y 4 filtros RXF2-5.

La máscara MCP 0 se configura para recibir mensajes de Broadcast(6) , es decir los que pueden recibir todos los módulos. Los filtros RXF0 y RXF1 se configuran para aceptar solo mensajes en donde PF = 240-255. La Máscara MCP 1 se configura para recibir mensajes globales o específicos. Los filtros de RXF3-5 se configuran para poder aceptar mensajes globales y el específico en el filtro RXF2.

c) Implementación:

En la función de inicialización J1939_init() se actualizan los filtros y las máscaras cuando el MCP2515 se encuentra en modo de configuración después de ser energizado.

```
MCP_Write( MCP_RXM0SIDH, 0x07 );                        // RXM0SIDH
MCP_Write( MCP_RXM0SIDL, 0x80 );                        // RXM0SIDL
MCP_Write( MCP_RXM1EID8, 0xFF );                        // RXM1EID8
MCP_Write( MCP_RXF0SIDH, 0x07 );                        // RXF0SIDH
MCP_Write( MCP_RXF0SIDL, 0x88 );                        // RXF0SIDL
MCP_Write( MCP_RXF1SIDH, 0x07 );                        // RXF1SIDH
MCP_Write( MCP_RXF1SIDL, 0x88 );                        // RXF1SIDL
MCP_Write( MCP_RXF2SIDL, 0x08 );                        // RXF2SIDL
MCP_Write( MCP_RXF2EID8, J1939_CA_ADDRESS );            // RXF2EID8
MCP_Write( MCP_RXF3SIDL, 0x08 );                        // RXF3SIDL
MCP_Write( MCP_RXF3EID8, J1939_GLOBAL_ADDRESS );        // RXF3EID8
MCP_Write( MCP_RXF4SIDL, 0x08 );                        // RXF4SIDL
MCP_Write( MCP_RXF4EID8, J1939_GLOBAL_ADDRESS );        // RXF4EID8
MCP_Write( MCP_RXF5SIDL, 0x08 );                        // RXF5SIDL
MCP_Write( MCP_RXF5EID8, J1939_GLOBAL_ADDRESS );        // RXF5EID8
```

Figura 42. Configuración de los Filtros y Mascaras de CAN del controlador MCP2515.

d) Verificación:

Los casos de prueba verifican:

-Que el modulo reciba mensajes específicos con su dirección destino e ignore los enviados a otros módulos.

-Reciba mensajes globales

Se muestran ejemplos del manejo de mensajes de J1939, un mensaje de grupo extendido es escrito por el Santo y recibido por la aplicación, un mensaje de destino específico en donde la dirección del sistema es 4C por lo que es procesado, un mensaje especifico con una dirección destino diferente 4B por lo que no es procesado y mensaje global en donde la dirección destino es 255 es procesado correctamente.

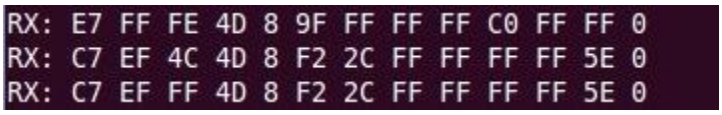


Figura 43. Mensajes procesados por la capa de Enlace de J1939

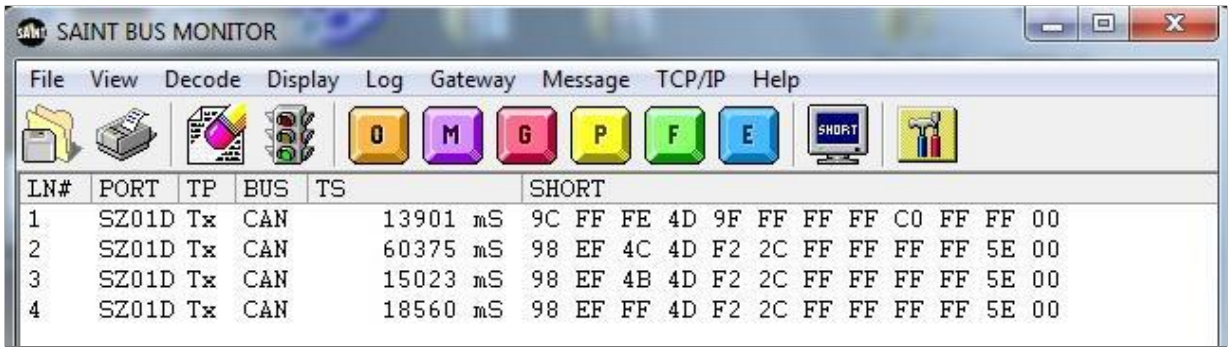


Figura 44. Mensajes procesados por la capa de Enlace de J1939 en el Santo.

El orden de los bytes en la aplicación se define:

Byte 1:PDUFormat_Top(3)
Byte 1:DataPage(1)
Byte 1:Res(1)
Byte 1:Priority(3)
Byte 2:PDUFormat
Byte 3:PDUSpecific
Byte 4:SourceAddress
Byte 5:DataLength
Byte 6-13: Data bytes

El santo despliega de manera distinta los Bytes 1 y 5 (formato y largo del mensaje). Se puede observar que el resto de los bytes que conforman la capa de enlace no presentan diferencias.

3.1.4 ETAPA 4. Capa de Transporte:

a) Requisitos:

La capa de transporte debe de proveer comunicación Punto a punto (CTS(14)/RTS(50)) y la capacidad de poder mandar 1785 bytes por mensaje.

Estructura y definición de los mensajes de TP

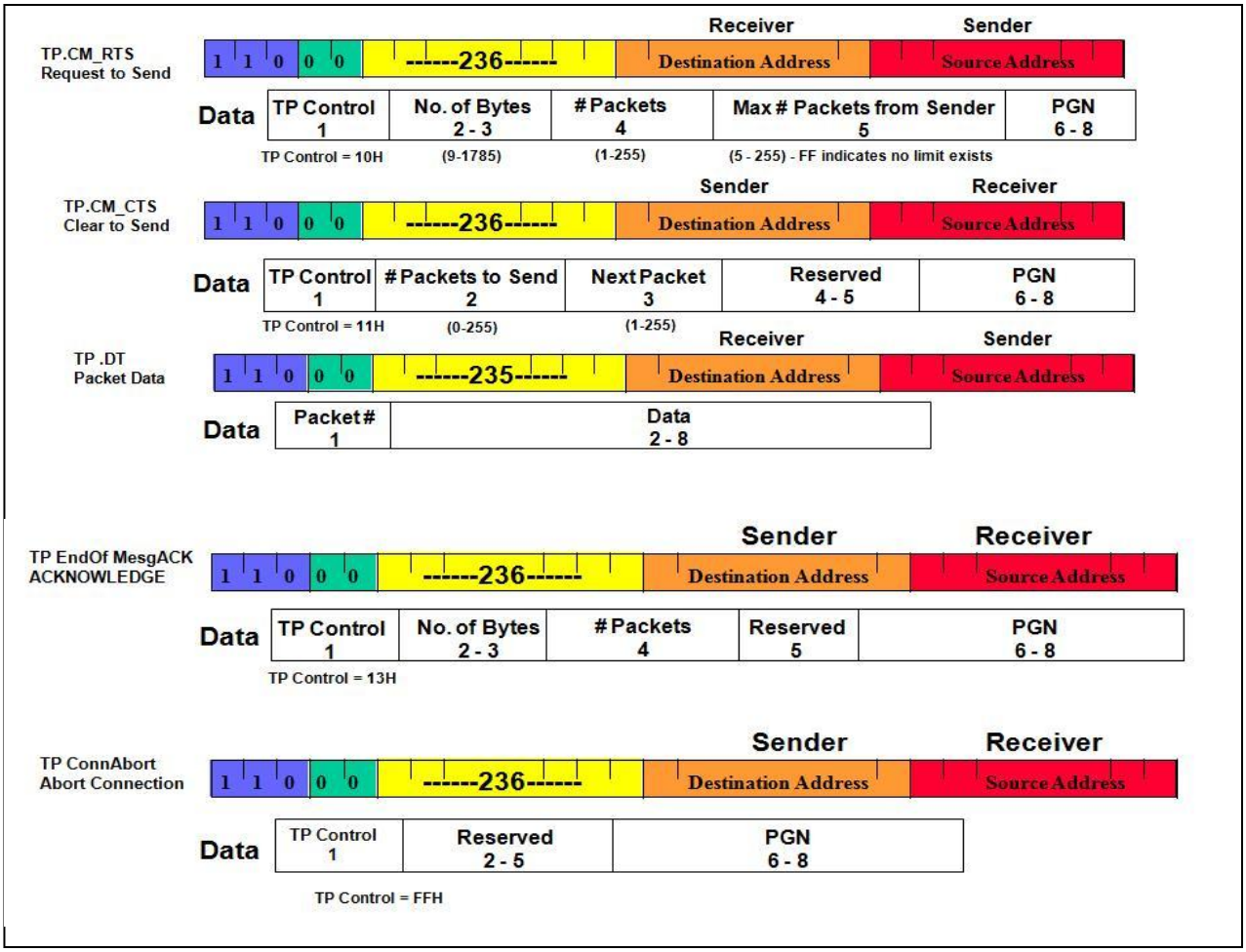


Figura 45. Estructura de los mensajes de la Capa de Transporte de J1939.

El modulo a enviar un mensaje comienza haciendo un requerimiento mediante el mensaje RTS, el modulo a recibir el mensaje debe de contestar con un mensaje CTS afirmando que puede comenzar a recibir los paquetes del mensaje. Una vez completado el mensaje, el modulo que recibió la información debe de contestar con un mensaje de ACK (Acknowledge) (1), si la información fue incompleta o no se cumplió alguno de los tiempos de espera especificados se enviara un mensaje de abortar. Tiempos para cerrar una conexión: 750 ms entre paquetes, 1250 ms después de mandar el mensaje CTS.

b) Diseño:

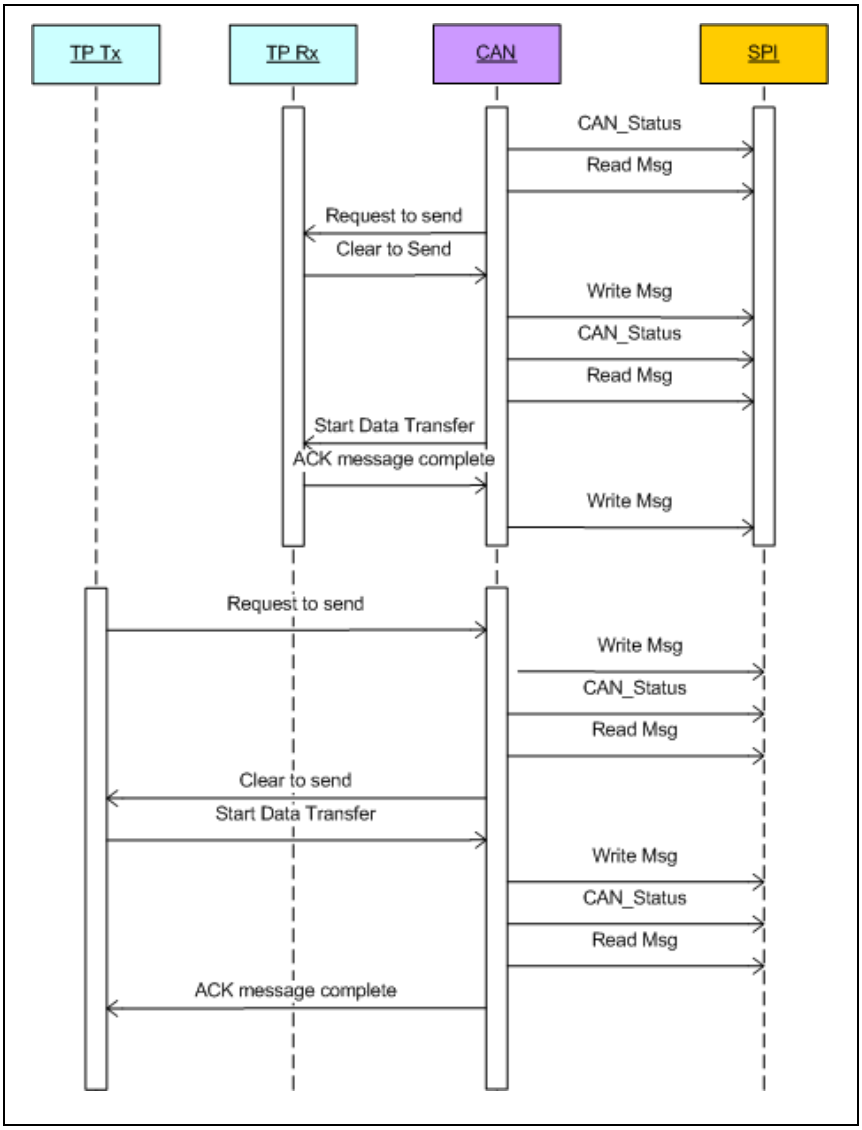


Figura 46. Diagrama de Secuencia UML de la capa de Transporte de J1939.

La sección de TP es dividida en 2 procesos transmisión y recepción. El proceso de CAN procesa solo mensajes simples es decir de 8 bytes. El diagrama engloba la transmisión de paquetes (varios mensajes sencillos) como las transiciones de transferencia de datos.

c) Implementación:

Se implementaron 2 máquinas de estados, una para recepción TP_Rx y otra para transmisión TP_Tx. Estas máquinas de estados son llamadas por el Handler de TP que pre filtra y actualiza los estados de las mismas. Cada máquina de estados detecta y actualiza los errores detectados en la transmisión y recepción de mensajes. En las figuras 47 y 48 se definen los diagramas de flujo que implementan estas funcionalidades.

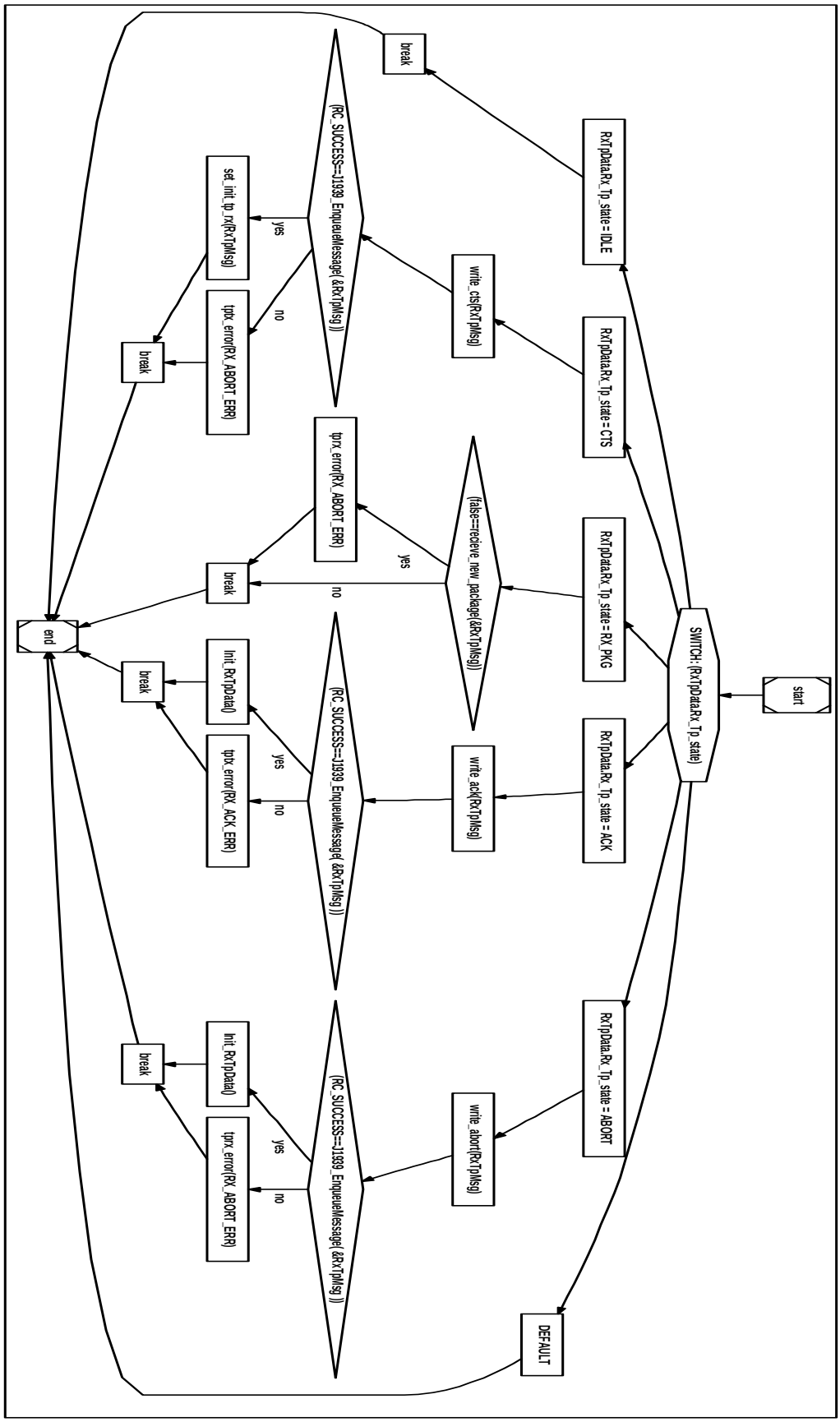


Figura 47. Implementación de la función de Transmisión de mensajes en la capa de transporte TP_Rx().

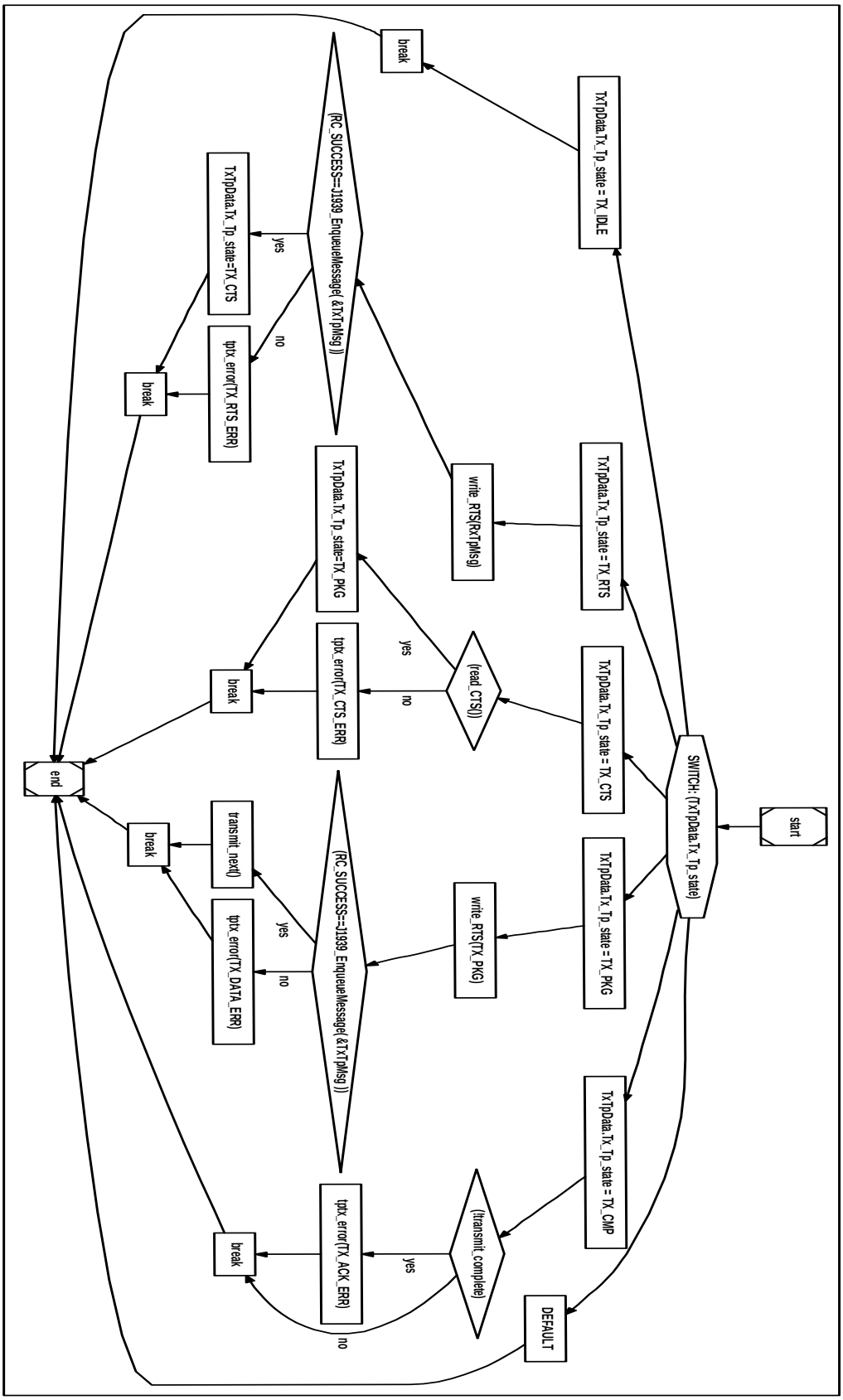


Figura 48. Implementación de la función de Recepción de mensajes en la capa de transporte TP_Tx().

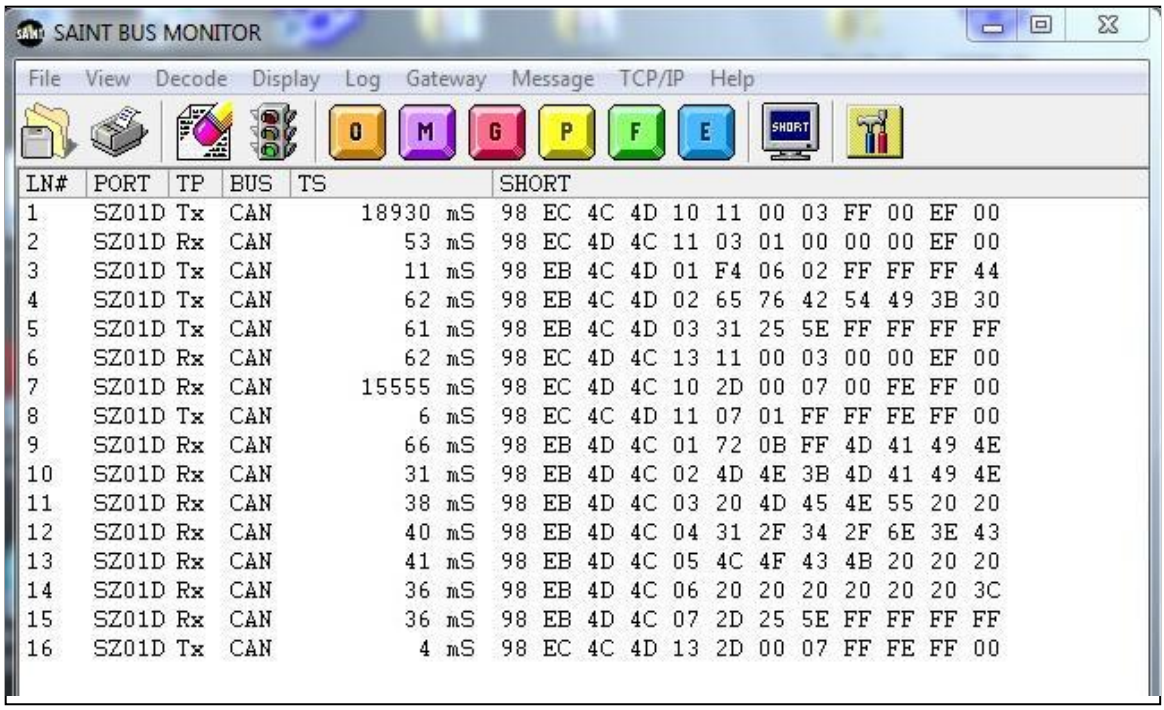
d) Verificación:

Para verificar la funcionalidad de esta etapa se utilizó el Santo como monitor de mensajes de CAN. En la Figura 49 y 50 se despliegan los mensajes de TP para la aplicación y el Santo respectivamente.

```
RX: C7 EC 4C 4D 8 10 11 0 3 FF 0 EF 0
RX: C7 EB 4C 4D 8 1 F4 6 2 FF FF FF 44
RX: C7 EB 4C 4D 8 2 65 76 42 54 49 3B 30
RX: C7 EB 4C 4D 8 3 31 25 5E FF FF FF FF
TP RX: 4D  0  EF  0  11  0  F4  6  2  FF  FF  FF  44  65  76  42  54  49  3B  30  31  25

TP TX: 4D  FE  FF  0  2D  0  72  B  FF  4D  41  49  4E  4D  4E  3B  4D  41  49  4E  20  4D  45  4E  55  2
0  20  31  2F  34  2F  6E  3E  43  4C  4F  43  4B  20  20  20  20  20  20  20  20  20  3C  2D  25  5E
RX: C7 EC 4C 4D 8 11 7 1 FF FF FE FF 0
RX: C7 EC 4C 4D 8 13 2D 0 7 FF FE FF 0
```

Figura 49. Despliegue de mensajes de la capa de Transporte de J1939.



LN#	PORT	TP	BUS	TS	SHORT
1	SZ01D	Tx	CAN	18930 mS	98 EC 4C 4D 10 11 00 03 FF 00 EF 00
2	SZ01D	Rx	CAN	53 mS	98 EC 4D 4C 11 03 01 00 00 00 EF 00
3	SZ01D	Tx	CAN	11 mS	98 EB 4C 4D 01 F4 06 02 FF FF FF 44
4	SZ01D	Tx	CAN	62 mS	98 EB 4C 4D 02 65 76 42 54 49 3B 30
5	SZ01D	Tx	CAN	61 mS	98 EB 4C 4D 03 31 25 5E FF FF FF FF
6	SZ01D	Rx	CAN	62 mS	98 EC 4D 4C 13 11 00 03 00 00 EF 00
7	SZ01D	Rx	CAN	15555 mS	98 EC 4D 4C 10 2D 00 07 00 FE FF 00
8	SZ01D	Tx	CAN	6 mS	98 EC 4C 4D 11 07 01 FF FF FE FF 00
9	SZ01D	Rx	CAN	66 mS	98 EB 4D 4C 01 72 0B FF 4D 41 49 4E
10	SZ01D	Rx	CAN	31 mS	98 EB 4D 4C 02 4D 4E 3B 4D 41 49 4E
11	SZ01D	Rx	CAN	38 mS	98 EB 4D 4C 03 20 4D 45 4E 55 20 20
12	SZ01D	Rx	CAN	40 mS	98 EB 4D 4C 04 31 2F 34 2F 6E 3E 43
13	SZ01D	Rx	CAN	41 mS	98 EB 4D 4C 05 4C 4F 43 4B 20 20 20
14	SZ01D	Rx	CAN	36 mS	98 EB 4D 4C 06 20 20 20 20 20 20 3C
15	SZ01D	Rx	CAN	36 mS	98 EB 4D 4C 07 2D 25 5E FF FF FF FF
16	SZ01D	Tx	CAN	4 mS	98 EC 4C 4D 13 2D 00 07 FF FE FF 00

Figura 50. Despliegue de mensajes de la capa de Transporte de J1939 en el Santo.

En la consola de la aplicación se observa el mensaje de TP recibido agrupado (TP RX), así como los paquetes que lo conforman (RX). Los datos coinciden con lo que escribe el Santo. La secuencia de RTS-CTS es ejecutada correctamente así como el mensaje ACK al final de la transmisión de paquetes. Para la transmisión de datos la aplicación muestra el mensaje transmitido agrupado (TP TX) así como la recepción de los mensajes CTS y ACK (RX). Los tiempos de lectura y escritura se encontraron dentro de rango.

3.1.5 ETAPA 5. Capa de Aplicación:

a) Requisitos:

La aplicación tiene como objetivo describir un ejemplo de uso práctico de la librería de J1939 desarrollada para este proyecto.

El menú de configuración y control definirá las siguientes opciones:

1. Habilitar la lectura y escritura
2. Deshabilitar lectura y escritura
3. Salir
4. Enviar Mensaje de TP
5. Enviar Mensaje
6. Cargar Mensaje a enviar
7. Cargar Mensaje de TP a enviar

La aplicación deberá de ser capaz de manejar y desplegar los errores detectados de CAN y de J1939:

Errores de SPI:

1) Puerto SPI no detecta datos de entrada

Errores de CAN detectados por MCP2515:

El controlador de CAN tiene la habilidad de detectar y manejar los errores de CAN detectados al momento de estar transmitiendo o recibiendo mensajes. Es posible detectar y desplegar los errores detectados mediante la lectura de un registro específico del controlador.

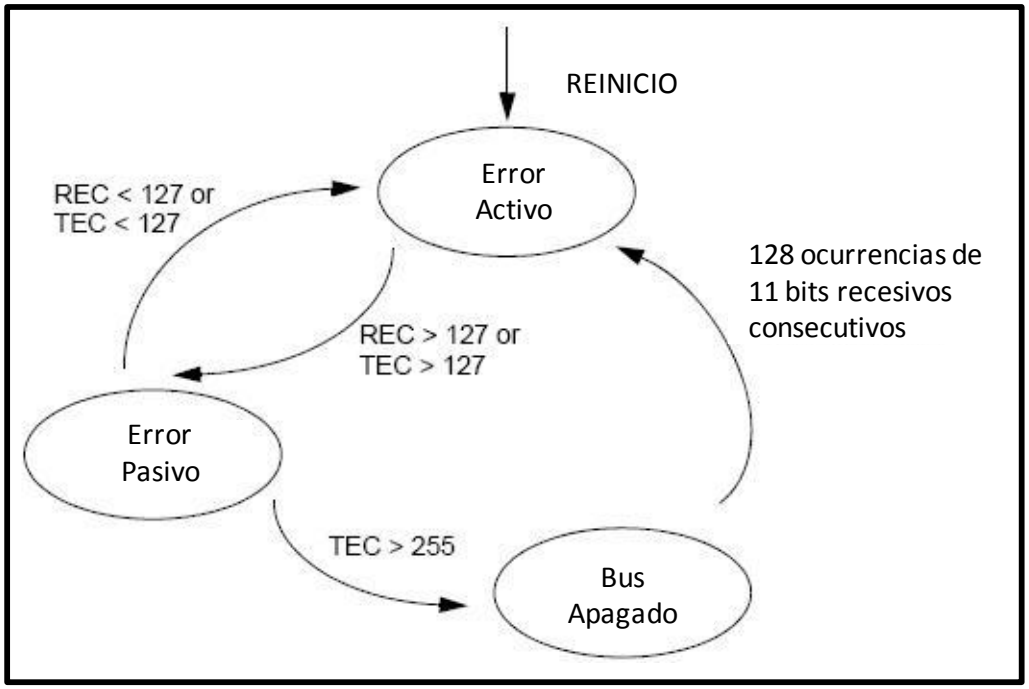


Figura 51. Estados de Error del MCP2515 [13].

Existen 3 estados de error posibles:

1) Error activo: Estado normal en donde se pueden transmitir mensajes sin restricción.
2) Error pasivo: pueden o no enviarse mensajes
3) Bus apagado: No se pueden enviar mensajes

Existen 2 contadores de errores REC (errores en la recepción), TEC (errores en la transmisión) Si el número de errores en TEC y en REC es menor de 128 el sistema se pone en Error Activo, si alguno de los 2 supera esta cuenta se pone en error pasivo y si el TEC supera los 255 el sistema apaga el BUS hasta recibir la secuencia de recuperación del sistema. Los tipos de error detectados de CAN: CRC Error (Cyclic redundancy check), Acknowledge Error (si el bit de Acknowledge no fue actualizado), Form Error (si se detecta un bit dominante al intentar transmitir, el mensaje debe ser reenviado), Bit Error (si el valor de algún bit ha sido cambiado), Stuff Error (Bit de stuff no detectado).Referirse al manual del MCP2515[13].

Registro con las banderas de errores:

Bit 7 Sobrecarga de recepción del buffer 1.

Bit 6 Sobrecarga de recepción del buffer 0.

Bit 5 Cuando se detectan 255 fallas sucesivas en el contador detector de errores de transmisión. Después de esto el bus se apaga como lo indica el mensaje.

Bit 4 TEC es mayor o igual a 128.

Bit 3 REC es mayor o igual a 128.

Bit 2 TEC es mayor o igual a 96 (a manera de advertencia).

Bit 1 REC es mayor o igual a 96 (a manera de advertencia).

Bit 0 TEC o REC son mayores a 96 (a manera de advertencia).

Errores de TP:

Los errores de TP son detectados en la capa de transporte de J1939 y pueden ser leídos y borrados desde la aplicación específica.

Al transmitir:

1) Recepción de mensaje CTS con dirección incorrecta
2) Recepción de mensaje CTS sin TP en proceso.
3) Recepción de mensaje de abortar.
4) RTS no puede ser enviado por tener mensaje en proceso.
5) CTS no recibido después de tiempo limite
6) Paquete con datos no puede ser enviado.
7) Respuesta de mensaje completado no recibido (ACK)
8) Mensaje de sobrecarga al llenarse el buffer de transmisión.

Al Recibir:

1) Mensaje Incompleto
2) Secuencia de mensajes recibida incompleta o errónea.

Con el fin de comprobar la interconexión con dispositivos utilizados actualmente en el mercado, la aplicación debe de ser capaz de recibir y desplegar mensajes provenientes de un RSA (Marca Delphi) utilizado en vehículos comerciales (Camiones) que implementan el protocolo J1939.

Figura 52. Módulo RSA.

El modulo debe ser capaz de conectarse y desplegar los mensajes que envía el RSA para cada uno de los controles, Encendido, volumen, banda, CD, Preset, +,-, TMSET, ALARM. Por el momento no es esperado realizar ninguna acción al recibir cada mensaje. Este control es muy similar al SWC Sistema de control en el volante del automóvil, por lo que para usos prácticos puede ser sustituido. Este componente provee la capacidad de controlar funciones de audio remotamente conectándose al radio del vehículo por medio de la red de CAN y utilizando el Stack de J1939.

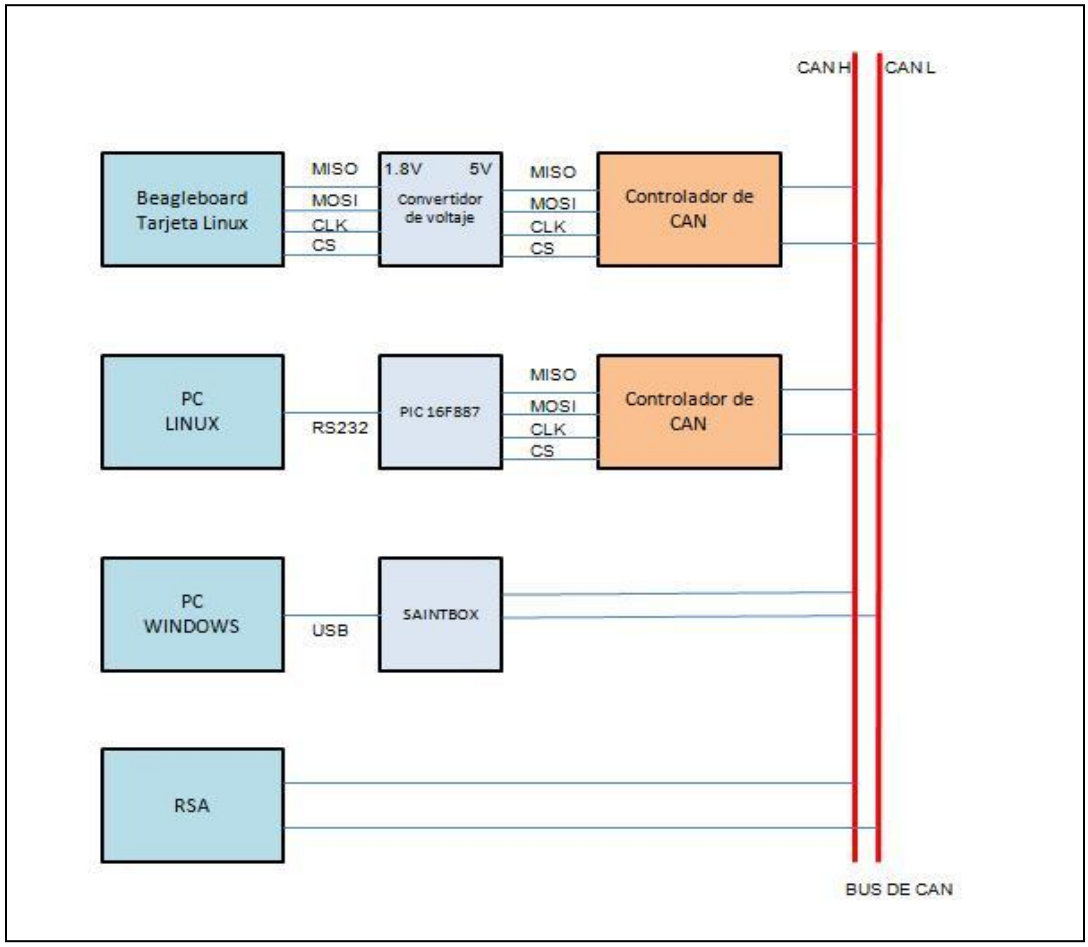


Figura 53. Conexión del sistema implementado de J1939.

b) Diseño:

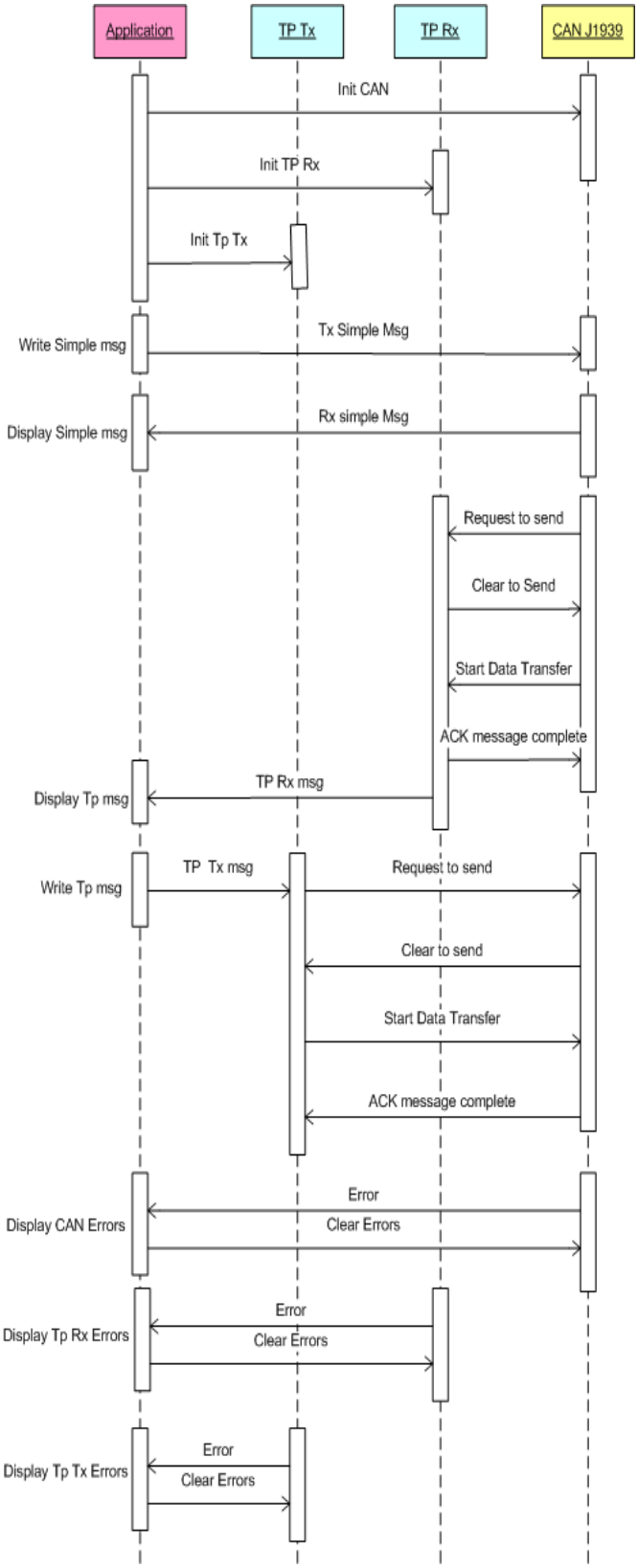


Figura 54. Diagrama de Secuencia UML de la Aplicación J1939.

El Diagrama muestra la interacción del proceso Aplicación con TP y CAN, desde su inicialización, transmisión, recepción y procesamiento de errores.

c) Implementación:

El primer diagrama de flujo implementa el hilo que llama a todos los procesos de J1939: Transmisión, Recepción de mensajes simples y de TP, así como el manejo de errores. El segundo diagrama implementa la función main en donde se inicializan las rutinas, se define la memoria compartida así como el semáforo y el hilo de J1939. También se especifican los modos de configuración y control de la aplicación principal.

La implementación del sistema utiliza un solo hilo para escribir y recibir mensajes simples de CAN así como de TP. Sin embargo podrían utilizarse de igual manera procesos separados para tener un sistema más rápido o eficiente. Sin olvidar considerar el manejo requerido de semáforos o mutex(41) para compartir memoria entre hilos o procesos. En esta aplicación los mensajes recibidos son enviados a otro proceso (aplicación grafica) mediante el uso de Memoria compartida utilizando la utilería del sistema POSIX (Portable Operating System Interface) (46) incluido en esta versión de Linux [10]. De igual manera fue requerido el uso de un semáforo (POSIX) para encolar mensajes desde el hilo principal hacia el hilo de J1939.

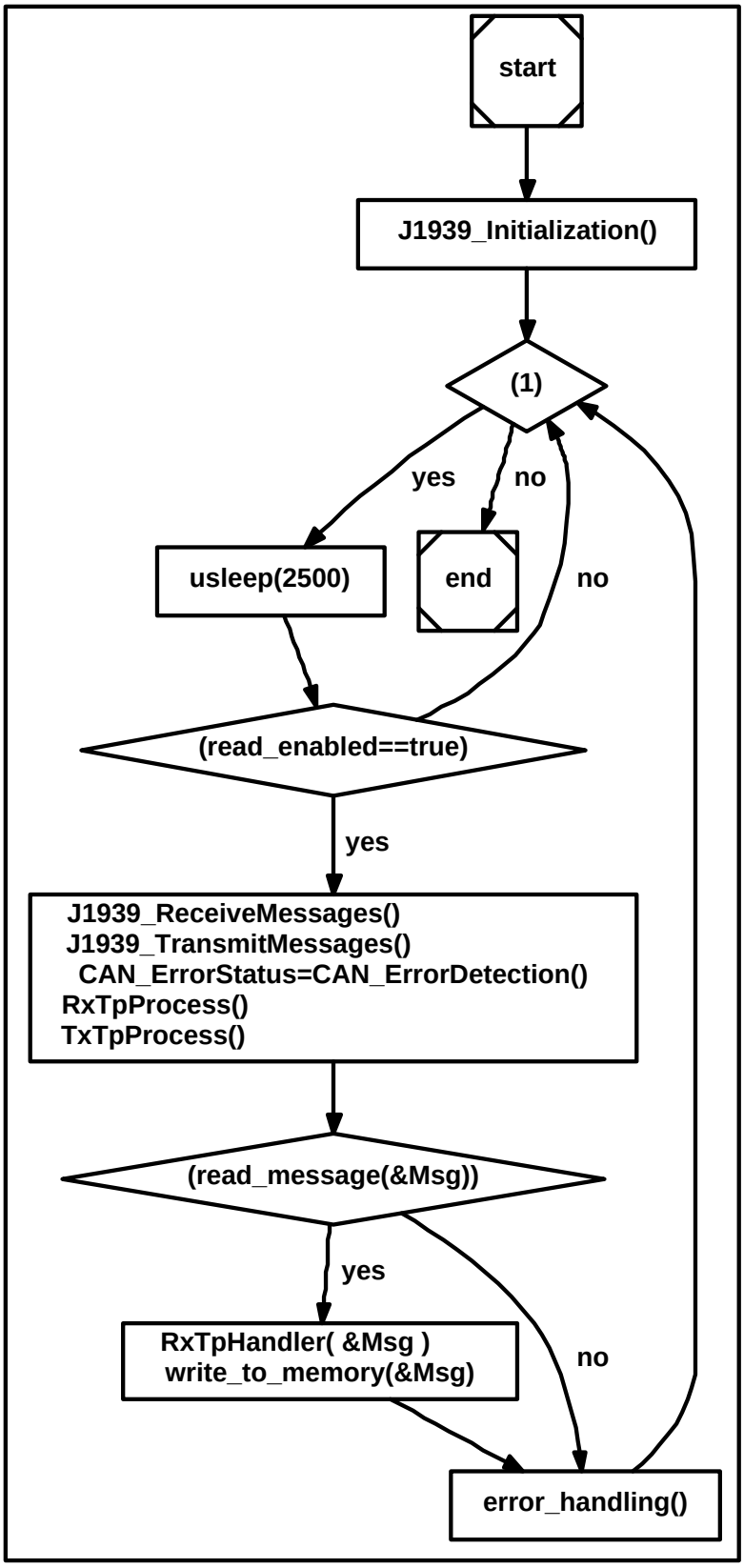


Figura 55. Implementación de la función que corre los eventos de transmisión y recepción de j1939 Thread com_cycle().

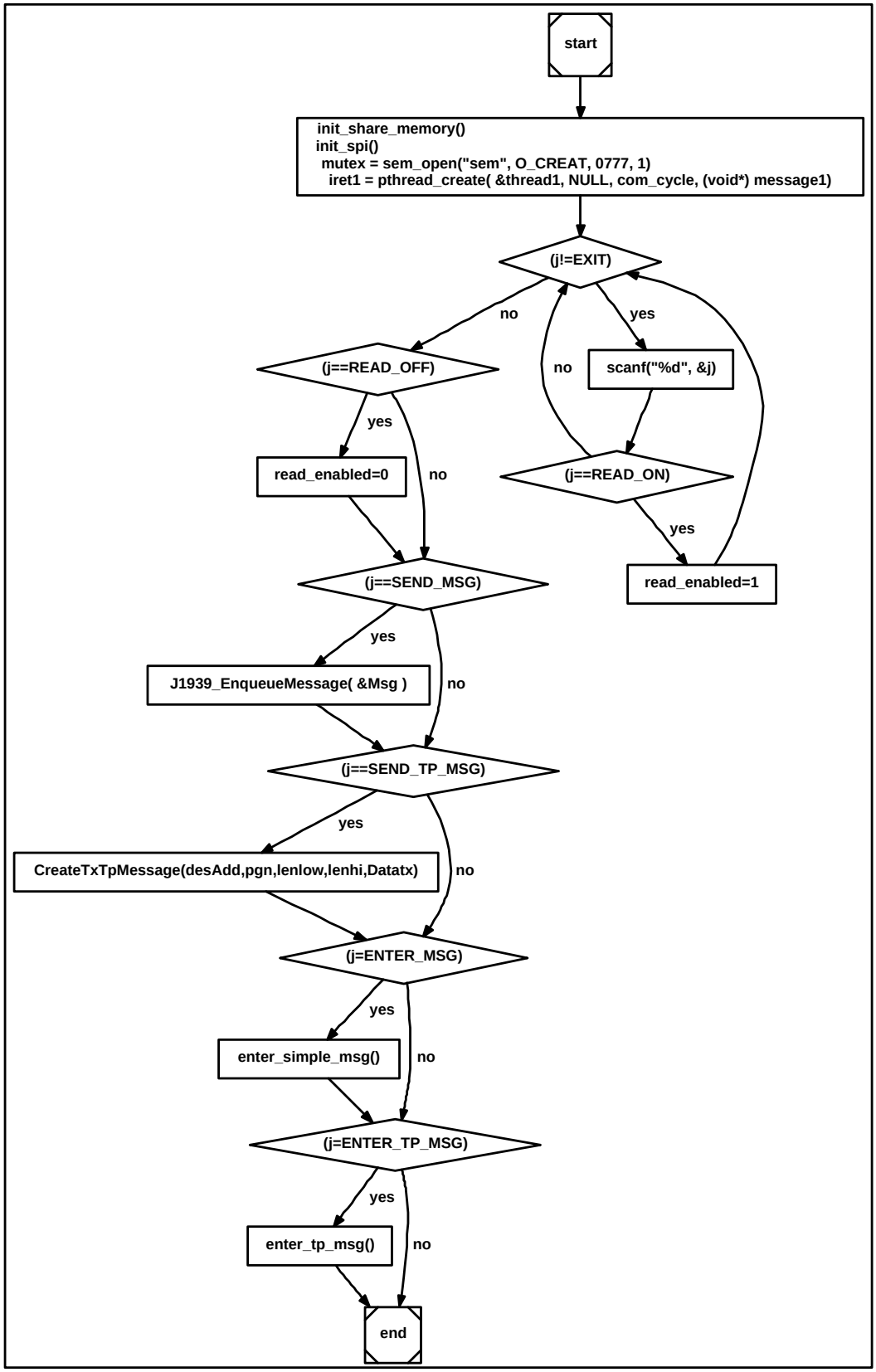


Figura 56. Implementación de la función principal main() de la capa de Aplicación de J1939.

d) Verificación:

Esta sección integran todas las etapas anteriores por lo que fue necesario repetir las pruebas mencionadas anteriormente así como los procesos de la detección de errores. Anexo a estas se verifico que el sistema fuera capaz de procesar y desplegar mensajes de un equipo RSA como se mencionó anteriormente.

La aplicación recibe y despliega los mensajes del RSA al presionar cada uno de sus botones de manera secuencial.

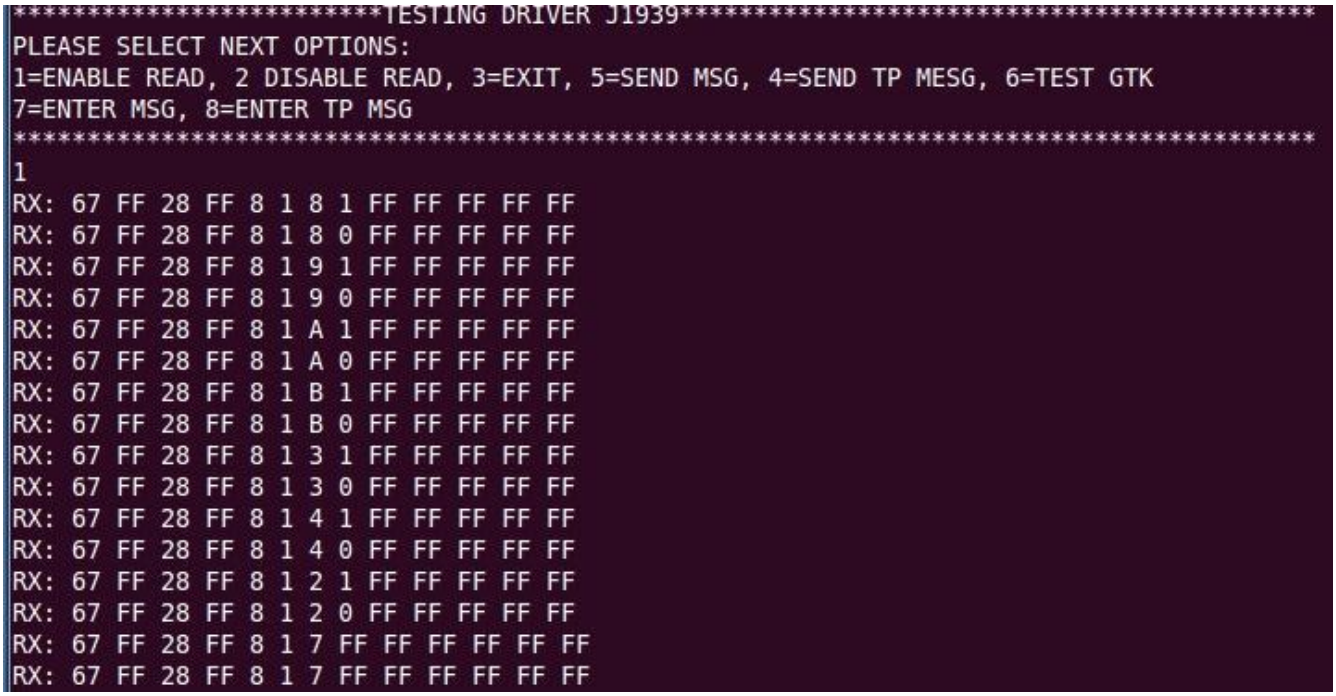
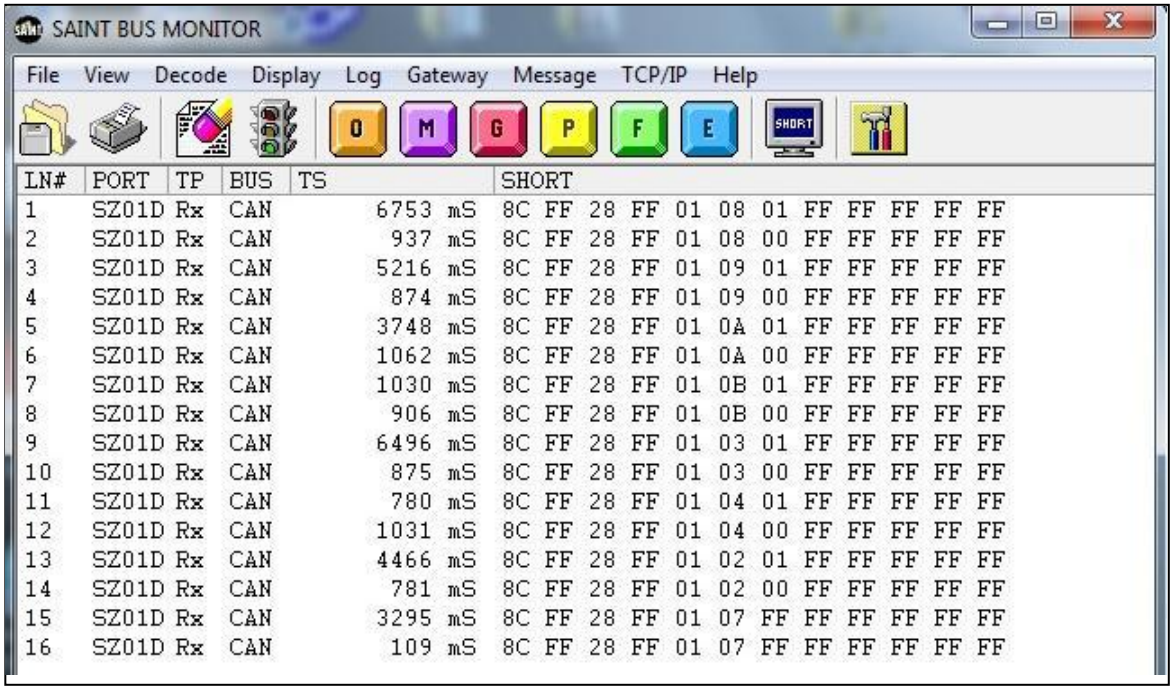


Figura 57. Mensajes recibidos por el RSA en la capa de Aplicación de J1939.

Como podemos observar el Santo recibe los mismos mensajes procesados por la aplicación.



LN#	PORT	TP	BUS	TS	SHORT
1	SZ01D	Rx	CAN	6753 mS	8C FF 28 FF 01 08 01 FF FF FF FF FF
2	SZ01D	Rx	CAN	937 mS	8C FF 28 FF 01 08 00 FF FF FF FF FF
3	SZ01D	Rx	CAN	5216 mS	8C FF 28 FF 01 09 01 FF FF FF FF FF
4	SZ01D	Rx	CAN	874 mS	8C FF 28 FF 01 09 00 FF FF FF FF FF
5	SZ01D	Rx	CAN	3748 mS	8C FF 28 FF 01 0A 01 FF FF FF FF FF
6	SZ01D	Rx	CAN	1062 mS	8C FF 28 FF 01 0A 00 FF FF FF FF FF
7	SZ01D	Rx	CAN	1030 mS	8C FF 28 FF 01 0B 01 FF FF FF FF FF
8	SZ01D	Rx	CAN	906 mS	8C FF 28 FF 01 0B 00 FF FF FF FF FF
9	SZ01D	Rx	CAN	6496 mS	8C FF 28 FF 01 03 01 FF FF FF FF FF
10	SZ01D	Rx	CAN	875 mS	8C FF 28 FF 01 03 00 FF FF FF FF FF
11	SZ01D	Rx	CAN	780 mS	8C FF 28 FF 01 04 01 FF FF FF FF FF
12	SZ01D	Rx	CAN	1031 mS	8C FF 28 FF 01 04 00 FF FF FF FF FF
13	SZ01D	Rx	CAN	4466 mS	8C FF 28 FF 01 02 01 FF FF FF FF FF
14	SZ01D	Rx	CAN	781 mS	8C FF 28 FF 01 02 00 FF FF FF FF FF
15	SZ01D	Rx	CAN	3295 mS	8C FF 28 FF 01 07 FF FF FF FF FF FF
16	SZ01D	Rx	CAN	109 mS	8C FF 28 FF 01 07 FF FF FF FF FF FF

Figura 58. Mensajes recibidos por el RSA en la capa de Aplicación de J1939 en el Santo.

Otra prueba fue verificar la interconectividad de la aplicación con una aplicación de ambiente grafico utilizando GTK(25) (Ver figura 59). Esta conectividad se logra utilizando memoria compartida como se explica en el diseño de esta etapa.

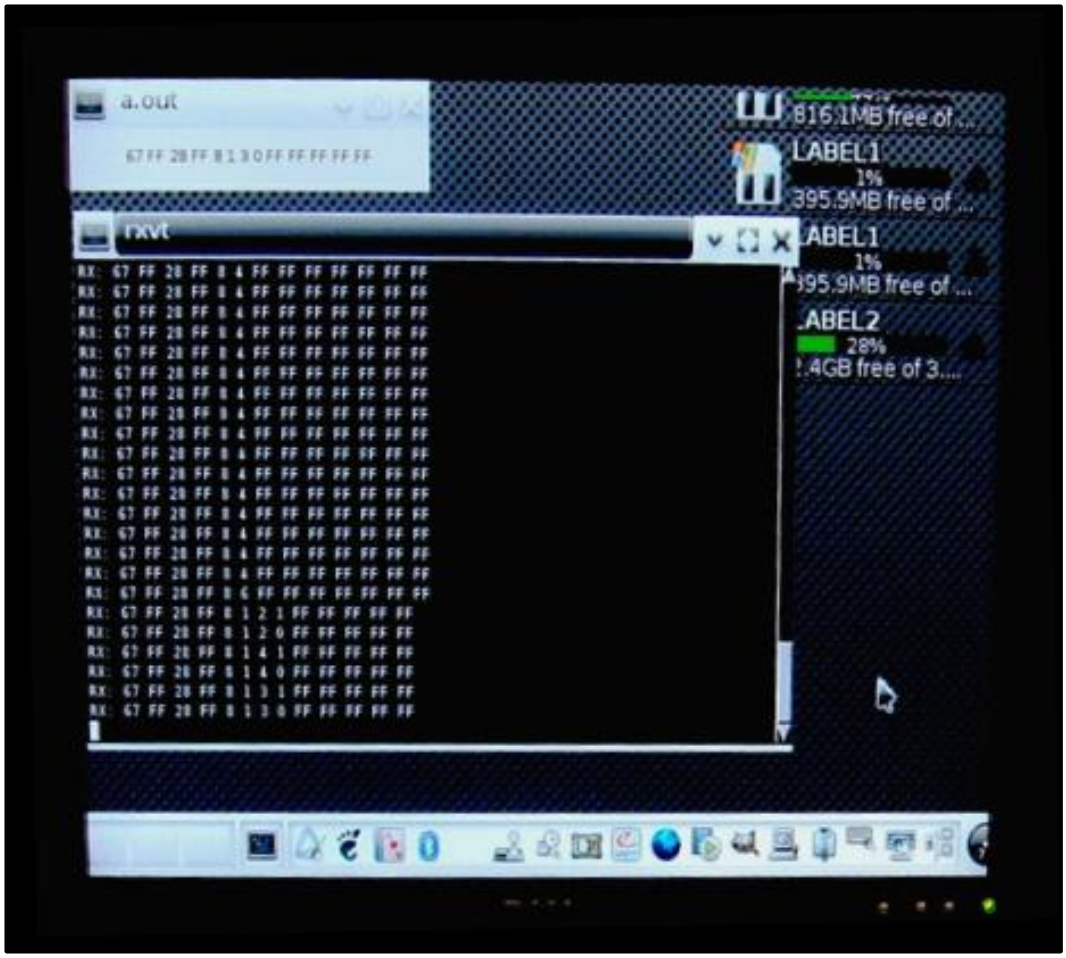


Figura 59. Recepción de mensajes J1939 con una aplicación con ambiente grafico de Linux (a.out).

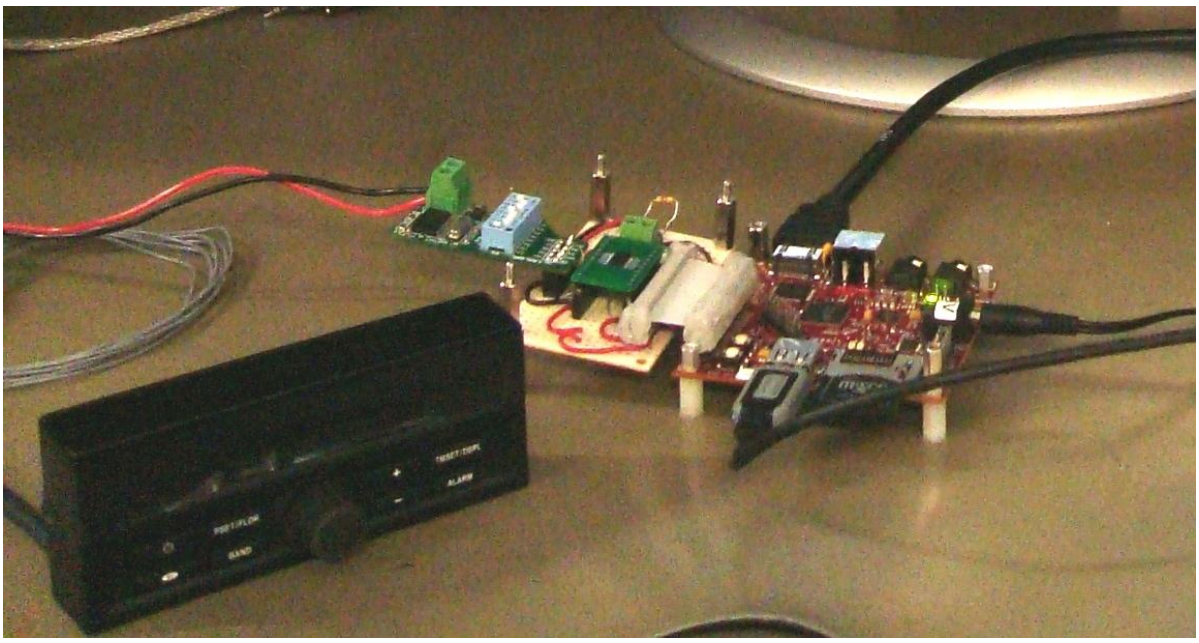


Figura 60. Sistema Completo: BeagleBoard, MCP2515 y RSA

Para medir la eficiencia de la aplicación se monitorearon las tareas en el sistema operativo utilizando la herramienta top de Linux obteniendo el resultado:

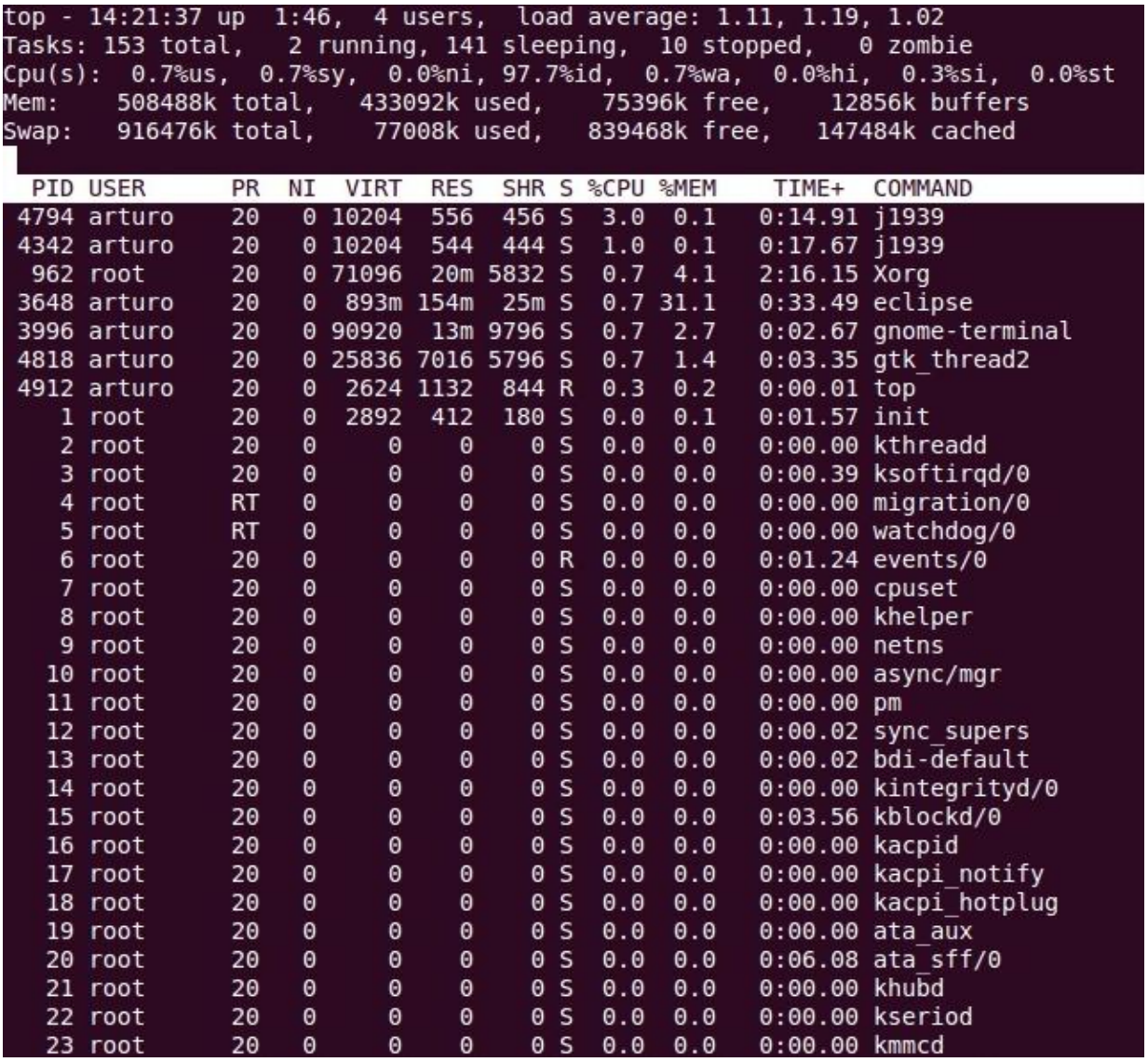


Figura 61. Análisis de la Capa de Aplicación de J1939 en sistema operativo Linux.

Podemos ver los 2 procesos de la aplicación J1939 con un porcentaje de uso del CPU del 4% y un 0.2% de uso de la memoria RAM.

Ancho de Banda y velocidad de procesamiento:

Como se comprobó en etapas anteriores tenemos que el Puerto SPI corre a 1MHz lo que nos permite lograr configurar el MCP2515 a los 250Kb/s requeridos por el protocolo J1939 (ver 3.1.2 ETAPA 2. Driver de CAN y SPI). En cuanto al procesamiento dependemos de la velocidad del hilo de recepción/transmisión de mensajes (ver implementación de función en Figura 55) que corre cada 2.5ms y de los dos de buffers de recepción de mensajes del controlador MCP2515. Por lo que el sistema puede procesar 2 mensajes cada 2.5ms, es decir un mensaje cada 1.25ms. Lo que supera las expectativas para esta aplicación dado que el RSA envía mensajes cada 5ms como máximo. Para mejorar aún más la velocidad de procesamiento se podría implementar el manejo de interrupciones desde el controlador de CAN y con esto evitar el poleo constante necesario para monitorear el estado de los mensajes.

4. CONCLUSIONES

4.1 Resultados:

El aporte más importante de este trabajo fue la implementación de una librería estática de simple instalación en sistemas Linux para PC y embebidos. Dicha librería permite configurar, controlar y monitorear las capas de Hardware, Enlace y de Transporte de conforman el protocolo J1939 para los controladores de CAN MCP2510/15.

Para demostrar la funcionalidad de la librería se desarrolló una aplicación de escritura y lectura de mensajes simples y de TP. En esta aplicación también demuestra cómo implementar un control IPC (Inter-process Communication)(31) para enviar los mensajes recibidos a otras aplicaciones.

Otro resultado importante fue la habilitación e implementación del driver de SPI para la tarjeta de desarrollo Beagleboard. Esto permitirá integrar más dispositivos al sistema como pueden ser cajas de Radio satelital, mecanismos de CD, pantalla táctiles, etc.

También se desarrolló una herramienta que permitió probar e implementar gran parte de la librería en una PC, lo que agilizó el trabajo, utilizando un Microcontrolador PIC que realizara la conversión y flujo de datos de UART a SPI. La librería es prácticamente la misma para PC excepto que en lugar de utilizar el puerto SPI utiliza el serial que se encuentra presente en la mayoría de las tarjetas de desarrollo con Linux embebido.

4.2 Desarrollos futuros:

Para mejorar la eficiencia de la librería actual se puede implementar el uso de interrupciones. El controlador de CAN puede detectar el estado de Recepción o Transmisión de mensajes para enviar la respectiva interrupción vía Hardware por puertos de entrada y salida. El OMAP puede recibir esta información y en base a ella disparar los eventos correspondientes eliminando el poleo actual.

Para continuar con el desarrollo de un sistema de multimedia para automóvil el siguiente paso será desarrollar una aplicación que reproduzca MP3(40). A su vez será necesaria una interfaz, probablemente con memoria compartida o mensajes IPC, para comunicar el reproductor con la aplicación o proceso que habilite J1939. La interfaz HMI puede ser desarrollada utilizando GTK, la aplicación de reproducción de audio y video puede

ser implementada utilizando Gstreamer(24), en donde será necesario validar el uso legal de sus licencias. En cuanto al Hardware es posible utilizar la Tarjeta Beagleboard como prototipo y después sustituirla por otra de uso comercial como lo ofrece la marca Gumstix para tener así un reproductor de MP3 de bajo costo y con una interfaz de CAN o J1939 de uso dentro de cualquier vehículo con este tipo de red.

4.3 Conclusiones:

La metodología incremental con la que se desarrollo la presente tesis permitió organizar y administrar el tiempo requerido para desarrollar un producto final confiable. Cada etapa de desarrollo incluyo una metodología de verificación que nos permitió un desarrollo acelerado de todas sus partes disminuyendo el número de problemas que implica una integración de componentes sin verificación previa.

Existen productos similares que tienen un costo y aun así requieren tiempo de desarrollo para adecuarse a la plataforma específica. Existen drivers de código abierto pero no consideran la arquitectura del Stack J1939 en su diseño original lo que requiere el desarrollo de parches que afectarían el desempeño del mismo. En esta tesis se presento el desarrollo de una librería de J1939 de fácil uso e instalación en sistemas Linux que requieran del controlador MCP2515 para integrar el protocolo de CAN.

El desarrollo de sistemas embebidos utilizando Linux puede ser en cierta medida más económica; sin embargo, no a corto plazo dado que debemos de considerar el tiempo requerido de capacitación e instalación de las herramientas de desarrollo. Este proceso puede ser muy largo y complicado dado que la información de soporte se encuentra muy dispersa y a veces no es muy confiable, haciendo que el tiempo de pruebas se incremente considerablemente. Es necesario considerar un proveedor de Hardware con experiencia en Software y que pueda presentar un plan de capacitación y soporte durante el desarrollo para acelerar aun mas este proceso.

A. APENDICE

A.1 Bibliografía:

[1] ABI Research's. In-Vehicle Networking Report (4Q 2004)
Abril, 2010
_www.abiresearch.com/home.jsp.

[2] Altera Corporation. Nios II Processor Reference
Febrero, 2011
_www.altera.com

[3] Andrew Wrigley .Steering-Integrated Driver Controls for Sunswift IV
The University of New South Wales
2009

[4] BeagleBoard.org. BeagleBoard System Reference Manual
2009
_www.BeagleBoard.org

[5] EE Times Embedded.com. Embedded Market Survey.
Abril, 2010
_www.eetimes.com

[6] Eric López Pérez. Protocolo SPI(Serial Peripherical Interface). Teoría y Aplicaciones Ingeniería en Microcontroladores
2009

[7] Gang Dai, Guanghua Gong, Beibei Shao, Wei Su. Design and implementation of multi-channel CAN Communication Interface based on Embedded Linux
2009
International Conference on Electronic Measurement & Instruments.
_www.ieeexplore.ieee.org/Xplore/guesthome.jsp?reload=true

[8] IP Provider CAST, Inc .CAN Bus Controller Core
Febrero, 2011
_www.cast-inc.com

[9] Merciadri Luca, Koen Kooi. Angstrom Manual
Diciembre, 2010
_www.angstrom-distribution.org

[10] Michael KerrisK. The Linux Programming Interface
2010
ISBN-10: 1-59327-220-0

[11] Michael O'Donnell. Automotive Telematics: Open-source automotive-grade Linux Is the future
Agosto 2005
_www.Embedded.com

[12] Michael Opdenacker, Thomas Petazzoni. Free Electrons Embedded Linux Kernel and driver development
Diciembre, 2010
_www.free-electrons.com

[13] Microchip Technology Inc . MCP2515 Stand-Alone CAN Controller With SPI Interface.
2007
_www.microchip.com

[14] Microchip Technology Inc. PIC16F882/883/884/886/887 Data Sheet
2008
_www.microchip.com

[15] MikroElektronika. CAN SPI Manual
2010
_www.mikroe.com

[16] P. Raghavan, Amol Lad, Sriram Neelakandan. Embedded Linux System Design and Development.
2006
ISBN -10: 0-8493-4058-6

[17] Peter Fellmeth, Thomas Löffler. Networking Heavy-Duty Vehicles Based on SAE J1939
Agosto, 2008
Technical Article
_www.vector.com/

[18] port.de . J1939 Protocol Stack
2009
_www.port.de

[19] SAE J1939 Standards Collection
Diciembre, 2010
_www.sae.org/standardsdev/groundvehicle/J1939a.htm

[20] Schach, Stephen. Object-Oriented and Classical Software Engineering. 7th ed. McGraw Hill Publishing Co.
2007.

[21] Semiconductors Bosch CAN protocol.
Diciembre, 2010
_www.semiconductors.bosch.de/en/20/can/1-about.asp

[22] Steve Blonstein, Alan Campbell. OMAP and DaVinci Software for Dummies
2009
ISBN 978-0-470-39522-6

[23] Texas Instruments Incorporated. Automotive Infotainment Guide
2008
_www.focus.ti.com/docs/solution/folders/print/472.html

[24] Texas Instruments Incorporated. 8-BIT BIDIRECTIONAL VOLTAGE-LEVEL TRANSLATOR FOR OPEN-DRAIN AND PUSH-PULL APPLICATIONS
2007
_www.ti.com

[25] Texas Instruments Incorporated. OMAP3530/25 Applications Processor
2009
_www.ti.com

[26] Tran Nguyen and Bao Anh. Real-Time Operating Systems for small microcontrollers
Octubre, 2009
Published by the IEEE Computer Society
_www.ieeexplore.ieee.org/Xplore/guesthome.jsp?reload=true

[27] Vector CANtech, Inc. Networking Heavy-Duty Vehicles Based on SAE J1939
2009
_www.vector-worldwide.com

[28] Writing a Linux SPI driver
_www.jumpnowtek.com/index.php?option=com_content&view=article&id=57&Itemid=62
Enero, 2011

[29] Xilinx Inc . MicroBlaze Soft Processor
Diciembre, 2010
_www.xilinx.com/tools/microblaze.htm

[30] Yu Jianfeng C, Jiang Tingbiao. Study of LAN Embedded NC system based on ARM and DSP. International Symposium on Intelligent Information Technology Application Workshops.
2009
Published by the IEEE Computer Society
_www.ieeexplore.ieee.org/Xplore/guesthome.jsp?reload=true

A.2 Glosario:

(1) ACK: (Acknowledge) mensaje que retroalimenta al transmisor de haber recibido los datos correctamente.

(2) ANSI-C: es un estándar publicado por el Instituto Nacional Estadounidense de Estándares (ANSI), para el lenguaje de programación C.

(3) AUTOSAR: (Automotive Open System ARchitecture) Arquitectura de Sistemas Abierta para el sector Automotriz.

(4) BIOS: (basic input/output system) Sistema basico de entrada/Salida.

(5) Bit-Rate: tasa de bits

(6) Broadcast Menssages: Mensajes de grupo extendido.

(7) Buffers: es una ubicación de la memoria en un Disco o en un instrumento digital reservada para el almacenamiento temporal de información digital.

(8) BUILDING BLOCK: bloque o unidad de construcción de software.

(9) CAN: (Controller Area Network) es un protocolo de comunicaciones desarrollado por la firma Robert Bosch GmbH, basado en una topología bus para la transmisión de mensajes en sistemas distribuidos, además ofrece una solución a la comunicación entre múltiples unidades centrales de proceso.

(10) can4linux: CAN driver para Linux

(11) CODECs: es la abreviatura de codificador-decodificador

(12) CRC: (Cyclic redundancy check) Comprobación de redundancia cíclica, un mecanismo de detección de errores en sistemas digitales

(13) Crosscompiler: Compilador que obtiene ejecutable de una arquitectura distinta al CPU en el que se genera. utilizado para compilar aplicaciones de Sistemas embebidos en PC´s convencionales.

(14) CTS: (clear to send) Respuesta del RTS

(15) DeviceNet: es un protocolo de comunicación usado en la industria de la automatización

(16) DMA: (Direct Memory Access) Acceso directo a Memoria

(17) DMIPS: Dhrystone MIPS (Million Instructions Per Second)

(18) Driver: Un controlador de dispositivo, llamado normalmente controlador (en inglés, device driver) es un programa informático que permite al sistema operativo interactuar con un periférico,

(19) DSP: (digital signal processor) Procesador de señales digitales

(20) DUAL CORE: Microprocesador multinúcleo (dos nucleos)

(21) ECUS: (Electronic Control Units). Unidad o nodo de comunicación en una red de CAN.

(22) Ethernet: estándar de redes de área local para computadoras

(23) FPGA: (Field Programmable Gate Array) es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada.

(24) Gstreamer: es un framework multimedia libre multiplataforma escrito en el lenguaje de programación C, usando la biblioteca GObject

(25) GTK: "GIMP Tool Kit" es una biblioteca del equipo GTK+, la cual contiene los objetos y funciones para crear la interfaz gráfica de usuario.

(26) Handhelds: (Handheld Computer, Handheld Device) Computadora portatil

(27) Handler: manejador controlador

(28) HMI: (Human Machine Interface) Interfaz de usuario.

(29) HUB: Concentrador, un dispositivo para compartir una red de datos o de puertos USB.

(30) Idle: El idle o Idle Time es la inactividad de un usuario o sistema

(31) IPC: (Inter-process Communication) comunicación entre procesos.

(32) ISDN: (Integrated Services Digital Network) Red Digital de Servicios Integrados

(33) KERNEL: (de la raíz germánica Kern) es el núcleo de un sistema operativo que implementa la abstracción del Hardware.

(34) LCD: (liquid crystal display) pantalla de cristal liquido

(35) LIN: (Local Interconnect Network) Estandar de comunicacion automotriz

(36) MISO: (Master Input Slave Output) Entrada del Maestro salida del Excalvo

(37) MMU: (Memory management unit). Unidad Controladora de Memoria.

(38) MOSI: (Master Output Slave Input) Salida del Maestro entrada del Exclavo

(39) MOST: (Media Oriented Systems Transport) define una red optolectronica que soporta transporte de datos multimedia para información y entretenimiento en el vehiculo de manera distribuida.

(40) MP3: formato de archivos de audio digital

(41) Mutex: algoritmos de exclusión mutua para proteger memoria o registros compartidos por distintos hilos o procesos

(42) MUX: (multiplexor de Entrandas y salidas)

(43) OEM: (original equipment manufacturer) fabricante de equipamiento original.

(44) OSI: (Open System Interconnection) El modelo de referencia de Interconexión de Sistemas Abiertos.

(45) PGN: (Parameter Group Number) Grupo de parametros que identifican a un mensaje de J1939

(46) POSIX: (Portable Operating System Interface) Interfaz de sistema operativo portable

(47) RISC: (reduced instruction set computer) Procesador con set de instrucciones reducido

(48) RSA: (Rear Seat Audio) Equipo que ofrece control remoto trasero en sistemas de audio automotriz

(49) RTOS: (Real Time Operating System) Sistema Operativo de Tiempo Real.

(50) RTS: (Request to send) Requerimiento para enviar

(51) SAE: Sociedad de Ingeniería Automotriz.

(52) SD: (Secure Digital) es un formato de tarjeta de memoria inventado por Panasonic

(53) SOC: (System on Chip). Sistema en un integrado.

(54) Soft Cores: Codigo que define la arquitectura de diseño de un procesador en un FPGA

(55) SPI: El Bus SPI (del inglés Serial Peripheral Interface) es un estándar de comunicaciones síncrono y bidireccional entre dispositivos electrónicos desarrollado por Motorola y ampliamente usado en la industria electrónica.

(56) Streaming: es la distribución de multimedia a través de una red de computadoras

(57) Stuff: Bit de stuffing

(58) Stuffing: proceso de agregar un Bit con polaridad invertida en una trama de 5 bits con la misma polaridad.

(59) SWC: (Steering Wheel Controls) es un controlador instalado en el volante de un vehiculo utilizado generalmente para acceder y controlar las funciones del radio del automóvil.

(60) Telemetría: permite la medición remota de magnitudes físicas y el posterior envío de la información hacia el usuario.

(61) TP: (Transport Protocol) Protocolo de la capa de transporte de J1939

(62) Transceiver: receptor transmisor, dispositivo en una red que puede adaptar señales de recepción y transmisión de datos.

(63) Watchdog: (perro guardián) es un mecanismo de seguridad que provoca un reset del sistema.

(64) x86: Arquitectura de procesadores (Intel)

B. Anexos

B.1 Configuracion de puertos Spi 3 y 4

```
static struct spi_board_info beagle_spi_board_info[] = {
        {
                .modalias       = "spidev",
                .max_speed_hz   = 48000000, //48 Mbps
                .bus_num        = 3,
                .chip_select    = 0,
                .mode = SPI_MODE_3 | SPI_CS_HIGH,
        },
};

static struct spi_board_info beagle_spi_board_info2[] = {
        {
                .modalias       = "spidev",
                .max_speed_hz   = 48000000, //48 Mbps
                .bus_num        = 4,
                .chip_select    = 0,
                .mode = SPI_MODE_0 | SPI_CS_HIGH,
        },
};
```

B.2. Configuracion de Puertos de entrada y salida

```
//McSPI3
OMAP3_MUX(SDMMC2_DAT3, OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI3_CS0
OMAP3_MUX(SDMMC2_DAT2, OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI3_CS1
OMAP3_MUX(SDMMC2_DAT0, OMAP_MUX_MODE1 | OMAP_PIN_INPUT),  //MCSPI3_SOMI (MASTER IN)
OMAP3_MUX(SDMMC2_CMD,  OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI3_SIMO (MASTER OUT)
OMAP3_MUX(SDMMC2_CLK,  OMAP_MUX_MODE1 | OMAP_PIN_INPUT), //MCSPI3_CKL
//McSPI4
OMAP3_MUX(MCBSP1_FSX,  OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI4_CS0 k26
OMAP3_MUX(MCBSP1_DX,   OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI4_SIMO (MASTER OUT) v21
OMAP3_MUX(MCBSP1_DR,   OMAP_MUX_MODE1 | OMAP_PIN_INPUT),  //MCSPI4_SOMI (MASTER IN) u21
OMAP3_MUX(MCBSP1_CLKR, OMAP_MUX_MODE1 | OMAP_PIN_INPUT),  //MCSPI4_CLK y21
```

B.3. Parche para incluir spidev driver en el Kernel de Linux

```
From c7c493b3b636746c13b58979d7ea5fe53a2f5605 Mon Sep 17 00:00:00 2001
From: arturo <arturo@ubuntu.(none)>
Date: Fri, 15 Apr 2011 09:39:42 -0700
Subject: [PATCH 2/2] spi-patch

---
 board-omap3beagle.c |   45 +++++++++++++++++++++++++++++++++++++++++++++
 1 files changed, 45 insertions(+), 0 deletions(-)

diff --git a/board-omap3beagle.c b/board-omap3beagle.c
index 664a9c6..b96614d 100644
--- a/board-omap3beagle.c
+++ b/board-omap3beagle.c
@@ -859,6 +859,25 @@ static struct spi_board_info beaglefpga_mcspi_board_info[] = {
                        .mode = SPI_MODE_1,
                },
 };
+static struct spi_board_info beagle_spi_board_info[] = {
+               {
+                       .modalias = "spidev",
+                       .max_speed_hz       = 48000000, //48 Mbps
+                       .bus_num = 3,
+                       .chip_select            = 0,
+                       .mode = SPI_MODE_3 | SPI_CS_HIGH,
+               },
+};
+
+static struct spi_board_info beagle_spi_board_info2[] = {
+               {
+                       .modalias = "spidev",
+                       .max_speed_hz       = 48000000, //48 Mbps
+                       .bus_num = 4,
+                       .chip_select            = 0,
+                       .mode = SPI_MODE_0 | SPI_CS_HIGH,
+               },
+};

 static void __init beaglefpga_init_spi(void)
 {
@@ -975,6 +994,28 @@ static struct omap_board_mux board_mux[] __initdata = {

                /* Camera - XCLK */
                OMAP3_MUX(CAM_XCLKA, OMAP_MUX_MODE0 | OMAP_PIN_OUTPUT),
+
+       //McSPI3
+
+       OMAP3_MUX(SDMMC2_DAT3, OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI3_CS0
+
+       OMAP3_MUX(SDMMC2_DAT2, OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI3_CS1
+
+       OMAP3_MUX(SDMMC2_DAT0, OMAP_MUX_MODE1 | OMAP_PIN_INPUT),  //MCSPI3_SOMI (MASTER IN)
+
+       OMAP3_MUX(SDMMC2_CMD,  OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI3_SIMO (MASTER OUT)
+
+       OMAP3_MUX(SDMMC2_CLK,  OMAP_MUX_MODE1 | OMAP_PIN_INPUT), //MCSPI3_CKL
+
+       //McSPI4
+
+       OMAP3_MUX(MCBSP1_FSX,  OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI4_CS0 k26
+
+       OMAP3_MUX(MCBSP1_DX,   OMAP_MUX_MODE1 | OMAP_PIN_OUTPUT), //MCSPI4_SIMO (MASTER OUT) v21
+
+       OMAP3_MUX(MCBSP1_DR,   OMAP_MUX_MODE1 | OMAP_PIN_INPUT),  //MCSPI4_SOMI (MASTER IN) u21
+
```

```
+       OMAP3_MUX(MCBSP1_CLKR, OMAP_MUX_MODE1 | OMAP_PIN_INPUT),  //MCSPI4_CLK y21

            { .reg_offset = OMAP_MUX_TERMINATOR },
 };
@@ -1079,6 +1120,10 @@ static void __init omap3_beagle_init(void)
            usb_ehci_init(&ehci_pdata);
            omap3beagle_flash_init();

+           printk(KERN_INFO "Beagle spidev ports registration");
+           spi_register_board_info(beagle_spi_board_info, ARRAY_SIZE(beagle_spi_board_info));
+           spi_register_board_info(beagle_spi_board_info2, ARRAY_SIZE(beagle_spi_board_info2));
+
            /* Ensure SDRC pins are mux'd for self-refresh */
            omap_mux_init_signal("sdrc_cke0", OMAP_PIN_OUTPUT);
            omap_mux_init_signal("sdrc_cke1", OMAP_PIN_OUTPUT);
--
            1.7.1
```

B.4. Lista de parches del Kernel en linux-omap-psp_2.6.32

```
SRC_URI_append_beagleboard = " file://logo_linux_clut224.ppm \
        file://beagleboard-xmc/0001-omap-Beagle-revision-detection.patch \
        file://beagleboard-xmc/0002-omap-Beagle-only-Cx-boards-use-pin-23-for-write-prot.patch \
        file://beagleboard-xmc/0003-omap-Beagle-no-gpio_wp-pin-connection-on-xM.patch \
        file://beagleboard-xmc/0004-omap3-beaglexm-fix-EHCI-power-up-GPIO-dir.patch \
        file://beagleboard-xmc/0005-omap3-beaglexm-fix-DVI-reset-GPIO.patch \
        file://beagleboard-xmc/0006-omap3-beaglexm-fix-power-on-of-DVI.patch \
        file://beagleboard-xmc/0007-beagleboard-hack-in-support-from-xM-rev-C.patch \
        file://beagleboard-xmc/0008-omap3-beagle-cleaned-up-board-revision-conditions.patch \
        file://beagleboard-xmc/0009-spidev-patch.patch \
```

B.5. Código de prueba spidev_test.c

```
static void transfer(int fd)
{
	int ret;
	uint8_t tx[] = {
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
		0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,
		0xF0, 0x0D,
	};
	uint8_t rx[ARRAY_SIZE(tx)] = {0, };
	struct spi_ioc_transfer tr = {
		.tx_buf = (unsigned long)tx,
		.rx_buf = (unsigned long)rx,
		.len = ARRAY_SIZE(tx),
		.delay_usecs = delay,
		.speed_hz = speed,
		.bits_per_word = bits,
	};

	ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
	if (ret < 1)
		pabort("can't send spi message");

	for (ret = 0; ret < ARRAY_SIZE(tx); ret++) {
		if (!(ret % 6))
			puts("");
		printf("%.2X ", rx[ret]);
	}
	puts("");
}

int main(int argc, char *argv[])
{
	int ret = 0;
	int fd;

	parse_opts(argc, argv);

	fd = open(device, O_RDWR);
	if (fd < 0)
		pabort("can't open device");

	/*
	 * spi mode
	 */
	ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
	if (ret == -1)
		pabort("can't set spi mode");

	ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
	if (ret == -1)
		pabort("can't get spi mode");

	/*
	 * bits per word
	 */
	ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
	if (ret == -1)
		pabort("can't set bits per word");
```

```
	ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
	if (ret == -1)
		pabort("can't get bits per word");

	/*
	 * max speed hz
	 */
	ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
	if (ret == -1)
		pabort("can't set max speed hz");

	ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
	if (ret == -1)
		pabort("can't get max speed hz");

	printf("spi mode: %d\n", mode);
	printf("bits per word: %d\n", bits);
	printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);

	transfer(fd);

	close(fd);

	return ret;
}
```