EMBEDDED SYSTEMS SPECIALIZATION PROGRAM

LIN Driver

SISTEMAS DE COMUNICACIÓN PARA SISTEMAS EMBEBIDOS E INTERNET DE LAS COSAS

Repositorio de la practica

https://github.com/DorianMtzP/Practica-4

Dorian Ricardo Martinez Preciado

Email: dorian.martinez@iteso.mx

Cesar Gomez Cruz

Email: cesar.gomez@iteso.mx

Josue Jahaziel Martinez Torres

Email: josue.martinez@iteso.mx

09/05/2023

Embedded Systems Specialization Program

www.sistemasembebidos.iteso.mx/alumnos

I T E S O A. C., Universidad Jesuita de Guadalajara

Periférico Sur Miguel Gómez Morín #8585, Tlaquepaque, Jalisco, México

Technical Report Number: ESE-O2014-001

1. Index

1. Index **2**
2. Introducción **3**
3 Desarrollo **5**
3.1 Transmision del header (Master) 5
3.1.1 Sync break (Master) 5
3.1.2 Sync (Master) 6
3.1.3 ID (Master) 6
3.2 Cálculo de bits de paridad. 7
3.3 Recepción de datos (Slave) 8
3.3.1 Sync break (Slave) 8
3.3.2 Sync y header (Slave) 8
3.3.3 Data (Slave) 9
3.3.4 Checksum (Slave) 10
3.4 Tarea Master 10
3.5 Call backs Esclavos 10
3.5.2 message_1_callback_local_slave 11
3.5.3 message_2_callback_local_slave 11
3.5.4 message_1_callback_slave 11
3.5.5 message_2_callback_slave 11
4 Resultados **12**
5 Conclusiónes **14**
5.1 Dorian R. Martinez Preciado 14
5.2 Josue J. Martinez 14
5.3 Cesar Gomez Cruz 14

2. Introducción

El protocolo Lin (Red de Interconexión Local) es un bus de sistema serial utilizado principalmente en la industria automotriz que cumple con la norma ISO-17987. Se trata de un protocolo de comunicación fiable y económico que permite a la computadora de a bordo del vehículo conversar con sus subsistemas.

Cualidades:

- Sistema de comunicación serial de bajo costo.
- Red SAE de clase A.
- Utilizado en la actualidad en vehículos comerciales.

Una red LIN se compone de un nodo maestro y varios nodos esclavos conectados a un Bus común.

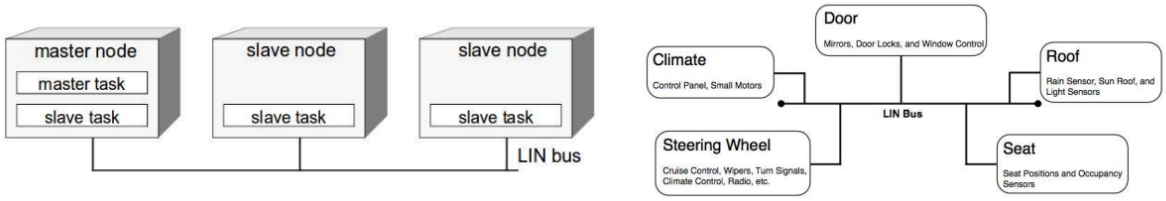

Figura 1.- Nodos de un sistema LIN.

Las redes LIN se utilizan para comunicar subsistemas, no críticos, tales como:

- Seguros de puertas.
- Motores de ventanillas.
- Luces interiores.
- Motores de asientos.

2.1 Requerimientos

Implementar una aplicación que utilice el protocolo LIN para comunicar dos o tres (dependiendo del número de integrantes del equipo) tarjetas K64:

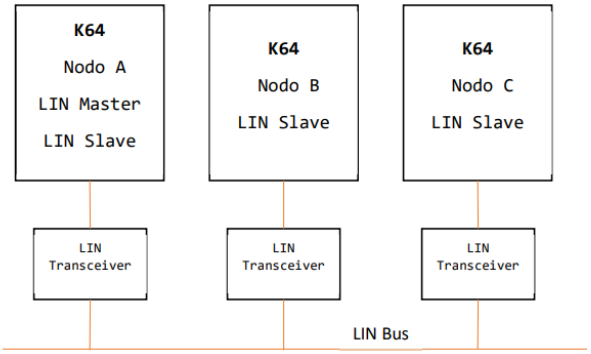

Figura 2. Nodos de LIN - K64F.

La aplicación debe implementar los siguientes mensajes de LIN.

ID	Contenido (Señales)	Slave transmisor	Slave receptor	Comportamiento
1	Color del LED: 2 bits 00 – OFF 01 – Rojo 10 – verde 11 - Azul	A	B y C	El LIN Master debe solicitar este mensaje cada segundo. El nodo que manda el response debe cambiar el valor de la señal cada vez que envíe el mensaje.
2	SW2.B: 1 bit (0 – Libre, 1 – Presionado) SW3.B: 1 bit (0 – Libre, 1 – Presionado)	B	A	El LIN Master debe solicitar este mensaje cada 100ms. El contenido depende del estado de los botones SW2 y SW3 en el nodo transmisor. El nodo receptor debe encender su LED rojo si uno de los botones esta presionado.
3	SW2.C: 1 bit (0 – Libre, 1 – Presionado) SW3.C: 1 bit (0 – Libre, 1 – Presionado)	C	A	El LIN Master debe solicitar este mensaje cada 100ms. El contenido depende del estado de los botones SW2 y SW3 en el nodo transmisor. El nodo receptor debe encender su LED rojo si uno de los botones esta presionado.

Figura 3. Señales a transmitir entre los nodos K64F.

3 Desarrollo

3.1 Transmision del header (Master)

En la generación de ***frames de mensajes*** (utilizadas para transmisiones normales de datos) se definió la siguiente estructura.

- Synch break.
- Synch.
- ID

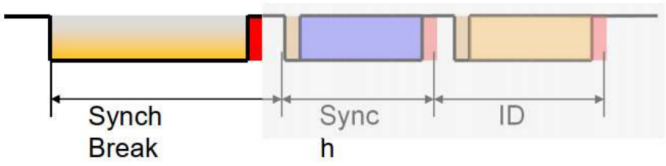

Figura 4. Frames de mensajes.

3.1.1 Sync break (Master)

El synch break debe estar en un estado dominante mínimo de trece bits lo que identifica el inicio del mensaje, para ello, de acuerdo con la hoja técnica o “datasheet” del microcontrolador FRMD-K64F **se deben cambiar los registros del UART Control Register `(UARTx_C2)` y del UART Status Register 2 `(UARTx_S2)` después de la inicialización del UART para activar un break signal y para determinar su longitud.**

Por lo que se añaden las siguientes líneas de código en la tarea del maestro:

```
/* Configure 13bit break transmission */
    handle->uart_rtos_handle->base->S2 |= (1<<UART_S2_BRK13_SHIFT);

    /* Send a Break It is just sending one byte 0, *** CHANGE THIS WITH A REAL SYNCH BREAK ****/
    //UART_RTOS_Send(handle->uart_rtos_handle, (uint8_t *)&synch_break_byte, 1);
/* Send the break signal */
    handle->uart_rtos_handle->base->C2 |= UART_C2_SBK_MASK;
    handle->uart_rtos_handle->base->C2 &= ~UART_C2_SBK_MASK;
```

Figura 5. Synch Break 13 bits - master_task

Posteriormente se transmite el Sync y el header.

3.1.2 Sync (Master)

Dentro del header se envía una sincronización adicional, la cual tiene un valor de 0x55 y sincroniza los relojes entre los nodos.

Consiste en un patrón de 8 bits, 0s y 1s alternados (0x55).

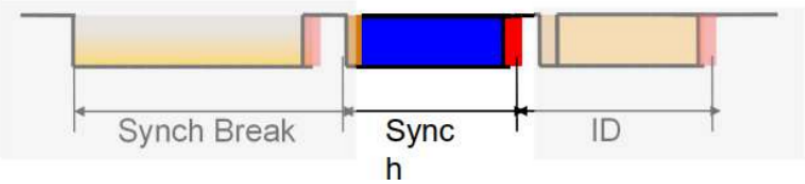

Figura 6. Sync de 0x55.

3.1.3 ID (Master)

El Header contiene el identificador del contenido y el tamaño del mensaje así como los bits de paridad.

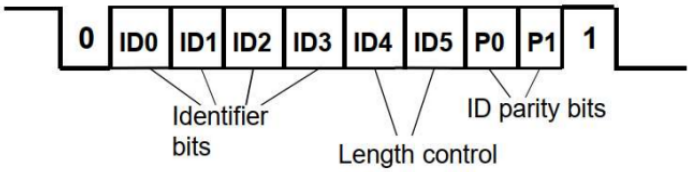

Figura 7. Campo de Identificación.

Los bits del ID0 - ID5 , proveen espacio para 64 identificadores, divididos en 4 subsets de mensajes, con tamaños de 2, 4 y 6 bytes.
Los bits P0 - P1, son los bits de paridad del identificador.

En la tarea del máster, antes de la transmisión del header, se calcula la paridad y se agrega a el header (bits P0 y P1) para que pueda ser transmitido.

```
/* Build and send the LIN Header */
/* Put the ID into the header */
lin1p3_header[1] = ID<<2;
/* TODO: put the parity bits */
parity = calculate_parity(lin1p3_header[1]);
lin1p3_header[1] = lin1p3_header[1] | parity;
```

Figura 8. Armado del header con bits de paridad en `master_task`.

3.2 Cálculo de bits de paridad.

En el cálculo y validación de los bits de paridad se realizaron operaciones XOR en el los ID bits, tanto los de identificación como los de longitud.

Para calcular los bits de paridad, se realizan las siguientes operaciones y el resultado serán dos bits, uno para el even y otro para el odd.

$$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4 \qquad \text{(even parity)}$$

$$P1 = \overline{ID1 \oplus ID3 \oplus ID4 \oplus ID5} \qquad \text{(odd parity)}$$

Figura 9. Ecuación para cálculo de paridad.

Se diseñó la función `calculate_parity` la cual calcula los bits en “even parity” y “odd parity”, la función recibe el header. Esta función mediante ciclos “`for`” aplica una XOR a los bits del ID necesarios para las operaciones y salta los bits que no son necesarios de acuerdo con la lógica de “even” / “odd”. Al final la función `calculate_parity()` regresa un uint8 con los bits de paridad.

```
uint8_t calculate_parity(uint8_t header){
        uint8_t bit = 0;
        uint8_t parityEv = 0;
        uint8_t parityOd = 0;

        //Calculate Even parity bit
    for (int i = 7; i > 2; i--){
        if(7 == i){
                bit = getBit(header,  i);
                parityEv = bit;

        }else if (i !=4){     //skip bit 4
            bit = getBit(header,  i);
            parityEv = parityEv^bit;
        }
    }
    //Calculate Odd parity
    for (int i = 6; i > 1; i--){

        if(6 == i){
                bit = getBit(header,  i);
                parityOd = bit;
        }else if (i !=5){    //skip bit 5
            bit = getBit(header,  i);
            parityOd = parityOd^bit;
        }
    }
    //PRINTF("even: %d\r\n", parityEv);
    //PRINTF("odd: %d\r\n", parityOd);
    return (parityEv << 1)|parityOd;

}
```

Figura 10. Funcion calculate_parity.

3.3 Recepción de datos (Slave)

Los dispositivos slave reciben los siguientes datos:

- Synch break.
- Synch.
- ID
- Data Frame (recibe o envía)
- Checksum (recibe o envía)

3.3.1 Sync break (Slave)

En la tarea del esclavo, se utilizan los registros del **`UARTx_S2 (Status register)`** para poder detectar el synch break de 13 bits. Se limpia la bandera de detección del sync break, se habilita el registro de détection de LIN Break y el micro se queda en espera de que esto suceda. Una vez que sucedió, se deshabilita la detección del break para que el driver de uart siga funcionando sin problemas.

Se añadieron las siguientes líneas de código en la tarea esclavo:

```
handle->uart_rtos_handle->base->S2 |= 0x01<<UART_S2_LBKDIF_SHIFT; //Clear the LIN Break Detect Interrupt Flag
handle->uart_rtos_handle->base->S2 |= 0x01<<UART_S2_LBKDE_SHIFT; //Enable LIN Break Detection
while((handle->uart_rtos_handle->base->S2 &  0x01<<UART_S2_LBKDIF_SHIFT) == 0x00) vTaskDelay(1); //Wait for the flag to be set
handle->uart_rtos_handle->base->S2 &= ~(0x01<<UART_S2_LBKDE_SHIFT); //Disable LIN Break Detection
handle->uart_rtos_handle->base->S2 |= 0x01<<UART_S2_LBKDIF_SHIFT; //Clear the LIN Break Detect Interrupt Flag
```

Figura 11. Detección de Synch break - slave_task

3.3.2 Sync y header (Slave)

Después de haber recibido un sync break se espera por el sync y el ID con la función `UART_RTOS_Receive`.

Para cumplir con los requerimientos, la tarea esclavo, implementa la detección de error e integridad de los mensajes:

- Detección del header
- Cálculo de los bits de paridad.
- Checksum de datos.

El primer bit después del sync break deve de ser un sync (0x55) si este no concuerda, se desecha ese mensaje, de lo contrario se analiza el header recibido (ID+len+parity) y se calcula y compara la paridad, si la paridad concuerda se procede a verificar si el ID existe en el dispositivo esclavo, de lo contrario se desecha y se vuelve a esperar por un sync break.

```
if(/*(lin1p3_header[0] != 0x00) &&*/
   (lin1p3_header[0] != 0x55)) {

    /* Header is not correct we are ignoring the header */
    continue;
}else
{
    PRINTF("Header: %x \r\n" , lin1p3_header[1]);
    /* TODO: Check ID parity bits */

    parity = calculate_parity(lin1p3_header[1]);

    if((parity & lin1p3_header[1]) != parity){
        continue; //Go to loop start
    }
```

Figura 12. Comprobando sync y paridad de header.

3.3.3 Data (Slave)

El esclavo, dependiendo de su configuración, puede enviar o recibir bytes de datos. En el driver esto se define mediante el esta variable `node_config.messageTable[1].rx = 0;` Si es 0 el esclavo envía los bytes de datos solicitados al bus, si es 1 el esclavo espera datos del bus.

La cantidad de datos a recibir o enviar es determinada por los bits de longitud del Header. Después de enviar los bytes de datos es necesario enviar un byte con el checksum del mensaje. En caso de estar recibiendo, el checksum debe de ser verificado antes de ejecutar el callback correspondiente al ID en cuestión. Si el checksum no coincide, el mensaje se desecha y se vuelve a esperar por un sync break.

```
if(handle->config.messageTable[msg_idx].rx == 0) {
    /*If the message is in the table call the message callback */
    /* User shall fill the message */
    handle->config.messageTable[msg_idx].handler((void*)lin1p3_message);
    /* TODO: Add the checksum to the message */
    //ONLY MESSAGE WITHOUT HEADER
    chsum = (uint8_t *)checksum(lin1p3_message , message_size);
    //PRINTF("chsum %x\n\r", chsum);

    /* Send the message data */
    UART_RTOS_Send(handle->uart_rtos_handle, (uint8_t *)lin1p3_message, message_size-1);
    UART_RTOS_Send(handle->uart_rtos_handle, &chsum, sizeof(chsum));
}
else {
    /* Wait for Response on the UART */
    UART_RTOS_Receive(handle->uart_rtos_handle, lin1p3_message, message_size, &n);
    /* TODO: Check the checksum on the message */
    chsum = checksum(lin1p3_message , n - 1);
    if(chsum != lin1p3_message[n-1]){
        continue;
    }
    /*If the message is in the table call the message callback */
    handle->config.messageTable[msg_idx].handler((void*)lin1p3_message);
}
```

Figura 13. Envío y recepción de datos.

3.3.4 Checksum (Slave)

El checksum es calculado utilizando la versión clásica de LIN 1.3, el cual suma los valores de los datos a enviar y cada vez que son mayores a 0x100, se le resta un 0xFF, una vez finalizada la sumatoria se invierte el resultado.

```
/*
 * Calculates Lin 1.3 Classic checksum
 */
uint8_t checksum(uint8_t message[] , uint8_t len){
    int checksum = 0;

    for(int i = 0; i< len - 1; i++){
        checksum += message[i];
        if(checksum >= 0x100)
            checksum -= 0xff;
    }
    //PRINTF("checksum: %x \r\n", ~((uint8_t)checksum));

    return 0xFF&(~((uint8_t)checksum));
}
```

Figura 14. Checksum

3.4 Tarea Master

La tarea del dispositivo LIN Maestro es simplemente orquestar el envío de los headers al bus. En este caso, existen 3 ID's, (1,2,3) todos con longitud de 2 bytes. Los ID 2 y 3 se envían cada 100 ms y el ID 1 se envía cada segundo. (Ver figura 3)

3.5 Call backs Esclavos

Se implementaron 4 funciones callback en total, 2 para el esclavo local y 2 para los nodos esclavos externos.

3.5.1 Tablas de Mensajes

Slave A

ID	Long msg	Callback	Direccion Mensaje
1	2	`message_1_callback_local_slave`	Transmisor
2	2	`message_2_callback_local_slave`	Receptor
3	2	`message_2_callback_local_slave`	Receptor

Tabla 1.- Slave A

Slave B

ID	Long msg	Callback	Direccion Mensaje
1	2	`message_1_callback_slave`	Receptor
2	2	`message_2_callback_slave`	Transmisor

Tabla 2.- Slave B

Slave C

ID	Long msg	Callback	Direccion Mensaje
1	2	`message_1_callback_slave`	Receptor
3	2	`message_2_callback_slave`	Transmisor

Tabla 3.- Slave C

3.5.2 message_1_callback_local_slave

Slave A - Transmisor
Envía un response con el status del LED y enciende el led correspondiente.

3.5.3 message_2_callback_local_slave

Slave A - Receptor
Recibe un mensaje de 2 bytes y determina cual(es) switches fueron presionados. Si alguno fue presionado enciende el LED Rojo, de lo contrario lo apaga.

3.5.4 message_1_callback_slave

Slave B o C - Receptor
Recibe un mensaje de 2 bytes y determina cuál LED está encendido. Si alguno fue presionado enciende el LED Rojo, de lo contrario lo apaga.

3.5.5 message_2_callback_slave

Slave B o C - Transmisor
Envía un mensaje de 2 bytes cuyo contenido indica cuál switch fue presionado.

4 Resultados

Antes de empezar con la transmisión de los mensajes, se valida el synch break , el synch y el header del protocolo LIN.

Esto fue monitoreado utilizando un analizador lógico junto con la K64F y con otra tarjeta conectada mediante un protoboard.

Figura 15. Maestro transmitiendo synch break y header.

Tanto la paridad y el checksum se comprobaron haciendo operaciones manuales de desplazamiento de bits con las operaciones XOR y con la suma de los valores que se transmitieron respectivamente.

Se juntaron todos los nodos, es decir las tarjetas FRMD-K64F, una maestra la cual contenía una versión del código de inicialización del UART en específico del maestro y las otras con la configuración de los esclavos.

Se comprueba mediante el analizador lógico el envío del header y ID del maestro al esclavo y la respuesta del esclavo a la solicitud del maestro con los bits de IDs, los datos y el checksum. En siguiente imagen se muestra el bus de LIN, en la parte izquierda está el header transmitido por el Master, se puede apreciar un sync break, seguido de un sync (decimal 85 -> 0x55) y un header (dec 111-> 0b0110 1111) con ID = 0110, len 11 (8 bytes) y paridad 11. En la parte inferior, se observa la respuesta del slave, el cual envía 8 bytes (Del 1 al 8) y el checksum (0x219) al final.

Figura 16. Transmisión y recepción entre maestro y esclavo.

A continuación se muestra una imagen del ID2 con su repuesta indicando que ambos switches estan libres.

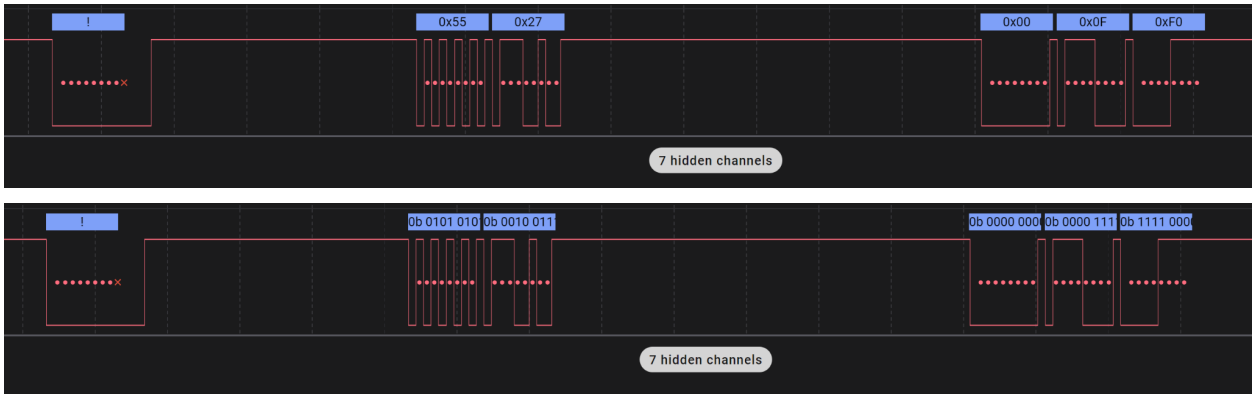

Figura 17. Transmisión y recepción entre maestro y esclavo. ID2

Por último, se prueban los mensajes de la figura 3, el LIN master solicitaba cada segundo el estado de los leds y cada 100ms de acuerdo a los botones del SW2 y SW3 de los nodos B y C (Dos esclavos, dos tarjetas K64F).

```
Slave B or C ID2 or 3 callback
SW2 Libre & SW3 Presionado
Header: 36
Header: 27
Slave B or C ID2 or 3 callback
SW2 & SW3 Libres
Header: 36
Header: 27
Slave B or C ID2 or 3 callback
SW2 & SW3 Libres
Header: 36
Header: 14
Header: 27
Slave B or C ID2 or 3 callback
SW2 & SW3 Libres
Header: 36
Header: 27
Slave B or C ID2 or 3 callback
SW2 & SW3 Presionados
Header: 36
Header: 27
```

Figura 17. Señales de los nodos de LIN.

La figura 17 corresponde a la impresión de mensajes de uno de los esclavos externos, el cual muestra el estado actual del nodo con su header y el estado de los botones, podemos ver que en el primer callback, el botón SW2 está libre pero el SW3 presionado, después se observa que el SW2 & SW3 están libres hasta que posteriormente ambos botones se presionan, validando cada uno de los estados. También se muestran cada uno de los header recibidos y cuales se ejecutan y cuáles se ignoran.

5 Conclusiónes

5.1 Dorian R. Martinez Preciado

Esta práctica fue muy divertida. Me hizo darme cuenta de que esto fue algo que pude haber usado hace como 8 años cuando hice un proyecto para el auditorio telmex en el que su sistema de alarma de incendios la cual imprimía por medio del puerto serial una cadena de caracteres que indicaba cual sensor se había activado. Mi forma de solucionarlo fue poner un máster que se encargaba de tener toda la interpretación y pasarsela al esclavo indicado. Si hubiera tenido más información sobre los buses la solución hubiera sido mucho más sencilla y la carga del CPU maestro hubiera sido mucho más sencilla. El haber hecho la parte de circuiteria creo que fue una buena adición a la práctica. Nosotros pedimos un transceiver pero el tiempo de entrega era muy elevado en méxico, terminamos pidiendo de USA y pero el costo de envio fue elevado, sería bueno considerar esto y quizas solicitar el material junto con el ESP32.

5.2 Josue J. Martinez

Esta práctica estuvo interesante, porque pudimos diseñar el driver con la funcionalidad completa desde el inicio hasta el final, considerando varios nodos, además que no solo transmitimos sino que revisamos la integridad de los mensajes, lo que permitió la implementación ser más robusta y aprueba de error, al leer la especificación de LIN, permitió que pudiéramos buscar en la hoja técnica de la tarjeta para poder diseñar el header, habilitando el synch break, el synch, una vez que entendimos cómo funcionaba el proyecto fue cuestión de diseñar los mensajes a transmitir y posteriormente revisar paridad y checksum, lo más difícil y en lo que batallamos fue la parte física, hacer el transceiver de LIN.

5.3 Cesar Gomez Cruz

La práctica resultó muy interesante, al permitirnos implantar un protocolo de comunicación ligero mediante bus con sistemas embebidos relativamente modestos. Esto tiene un potencial enorme para implantar proyectos que involucren redes de sensores y actuadores, con un maestro central que arbitre el tráfico (un paradigma muy común en redes industriales).
El armado del transceiver fue la parte que mayores complicaciones presentó, debido al tiempo requerido para adquirir los circuitos integrados y todos los componentes electrónicos. Al final optamos por implantarlo con un par de transistores 2N2222 para obtener la polaridad correcta.
Me parece que a estos protocolos, los automotrices, se les debe dedicar mayor tiempo de clase y hacer un par de ejercicios más extensos con Canoe.