Implementación de una red de comunicación y control para instrumentos de un laboratorio de técnicas analíticas nucleares

Eduardo Cunya[1,*], Óscar Baltuano[2], Patricia Bedregal[2]

[1] División de Materiales, INDE, Instituto Peruano de Energía Nuclear, Apartado 1687, Lima 41, Lima, Perú

[2] Dirección de Investigación y Desarrollo, INDE, Instituto Peruano de Energía Nuclear, Av. Canadá 1470, Lima 41, Perú

Resumen

En este artículo se describe la implementación de una red de comunicación y control para instrumentos de un laboratorio convencional y de procesos analíticos nucleares, basadas en el bus de campo para el control de dispositivos y máquinas CANOpen. Se presentan los componentes de hardware y software desarrollados así como las herramientas de instalación y configuración para la incorporación de nuevos instrumentos a la red.

Abstract

This paper describes the implementation of a communication network and control for a conventional laboratory instruments and nuclear analytical processes based on CANopen fieldbus to control devices and machines. Hardware components and software developed as well as installation and configuration tools for incorporating new instruments to the network are presented.

1. Introducción

El presente informe indica el procedimiento para elaborar un dispositivo de entrada/salida de tipo genérico que forma parte de una red sumergida [1,2] (embedded networks), el uso de Objetos de Datos de Procesos (transmisión y asignación), el uso de variables de retención en un Diccionario de Objetos, crear un programa propio para dispositivos inteligentes, usar mensajes CAN [3] personalizados para un dispositivo maestro encargado de la administración de la red (NMT), usar mensajes de emergencia para errores de usuario, etc. Toda la funcionalidad provista está contenida en la biblioteca de funciones de fuente abierta *CANopenNode*.

2. Experimental

El desarrollo experimental de la red de comunicación y control se realizó en una tarjeta de circuito impreso *ad-hoc*, que incluye el microcontrolador PIC 18F548 del fabricante Microchip, un circuito transmisor/receptor de señales sobre par trenzado de hilos conductores (transceptor) para CAN, el MCP 2551 de Microchip, indicadores luminosos (LED´s) del estado de operación del bus de comunicación y botones. La red también tiene un oscilador de 20 MHz para operar sobre el PIC (Figura 1). Para cualquier otra configuración o modo de operación del hardware, el archivo *CO_driver.h* tiene que ser modificado.

En esta etapa inicial solo se desarrolló el software de comunicación para un dispositivo sensor con ayuda de algunas herramientas de acceso libre en Internet (Licencia GNU) [4]; el cual comprende la depuración, simulación y elaboración del firmware desplegado en el PIC, utilizando la Biblioteca *CANopenNode* de la capa de aplicación conforme al estándar CANopen, el cual permite interconectar a la red dispositivos o instrumentos de diferentes fabricantes o desarrolladores.

2.1 Procedimiento

2.1.1 Diseño de Software

La construcción del software está basada en la pila de software de fuente abierta *CANopenNode* [5] elaborada por Janez Paternoster (janez.paternoster@siol.net). Priorizando la recepción de mensajes de datos antes que la transmisión a través de módulos de interrupción de alta y baja prioridad, respectivamente e implementando además la funcionalidad de un instrumento en la red con ayuda de una estructura llamada

* Correspondencia autor: ecunya@ipen.gob.pe

Diccionario de Objetos.

También se muestra como crear un dispositivo de entrada/salida genérico, el uso del objeto Datos de Proceso, el uso de variables persistentes en un Diccionario de Objetos, la creación del programa de usuario para un nodo maestro NMT, el uso de mensajes de emergencia para los errores de usuario durante la comunicación de datos y los mensajes acerca de la actividad de un nodo sobre la red, entre otros.

2.1.2 Requerimientos

1. Código fuente de la biblioteca CANopenNode versión 1.10
2. Entorno de programación MPLAB IDE de Microchip.
3. El compilador de Lenguaje C MPLAB C18 V3.00 o superior de Microchip.
4. Dos o más tarjetas con el microcontrolador PIC18F458 (u otro PIC 18F con CAN) y un transmisor/receptor CAN.
5. Un programador para dispositivos PIC (MPLAB ICD2 de Microchip).
6. Conocimiento sobre uso de MPLAB IDE y de MPLAB C18.
7. Conocimiento del protocolo de comunicación industrial CANopen.
8. Adaptador USB para conexión de PC a bus CAN.

2.2 Descripción

La red está compuesta por un conjunto de tres balanzas electrónicas de precisión, ubicadas en una sala distante a 40 m aproximadamente del ambiente donde se ubica la PC encargada del registro y almacenamiento de las lecturas de cada balanza (nodo maestro). Para las pruebas iniciales se dispusieron de solo dos dispositivos CANopen:

1. **Sensor:** balanza electrónica autónoma:
- Está basado en el perfil de dispositivo de Entrada/Salida genérico, según las especificaciones CAN in Automation [6,7] DS301 y DS401.
- Transmite lecturas de pesado en cada cambio de estado y de manera periódica con ayuda de un temporizador de eventos (TPDO 1).
- Produce señales de actividad sobre la red cada segundo (Latidos).

2. **Unidad de recepción de datos**: PC encargada del monitoreo del nodo sensor (balanza), registro y almacenamiento de lecturas de pesado:
- Recibe la lectura de pesado desde el Sensor (RPDO 0).
- Produce Latidos cada segundo.
- Monitorea los Latidos del Sensor.

Junto a los objetos de comunicación descritos anteriormente, cada nodo también usa los Objetos de Servicio de Datos, los objetos de comunicación de Emergencia y los objetos NMT (administración de la red).

2.3 Hardware utilizado

La circuitería está compuesta por un microcontrolador PIC18F458 de Microchip, un transmisor/receptor CAN, tres diodos LED's indicadores de estado de operación del bus y un boton de reposición (Figura 1). El transmisor/receptor CAN MCP 2551 de Microchip o el PCA82C250 de Philips pueden ser usados (Figura 2). Se utiliza un oscilador a cristal de 20 MHz para la frecuencia de reloj de operación del controlador. Si otra frecuencia de oscilador u otros pines para los diodos CAN son utilizados, el archivo *CO_driver.h* tiene que ser editado.

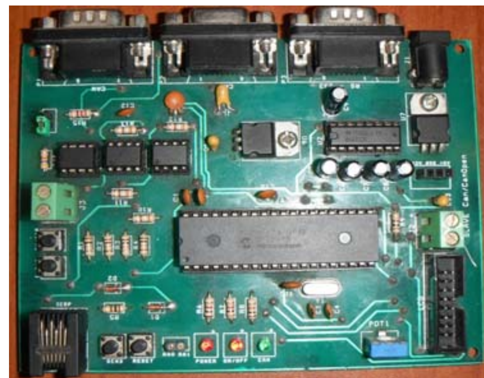


Figura 1. Tarjeta de comunicación elaborada para la aplicación.

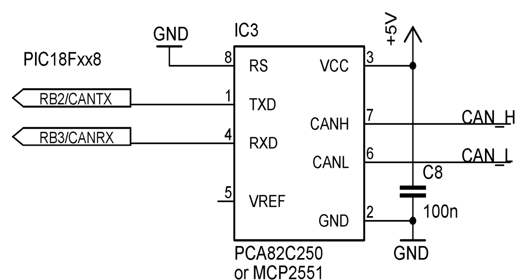


Figura 2. Transmisor/Receptor CAN conectado a PIC.

Descripción de los nodos de la red conforme al estándar DS303 [8] (Figura 3):

1. **Sensor:**

- Diodo LED de color verde entre las patillas RC3 y GND (LED de ejecución CAN).
- Diodo LED de color rojo entre las patillas RC5 y GND (LED de error CAN).
- Trasmisor/Receptor RS232 con las líneas del puerto de comunicación serial, patillas RC7 y RC6.

2. **Unidad de recepción de datos**:

- Indicador de ejecución de red CAN en programa de usuario de nodo maestro (PC).
- Indicador de error de red CAN en programa de usuario de nodo maestro.
- Gestión de archivo de datos (lectura de pesado) en programa de usuario de nodo maestro (PC).

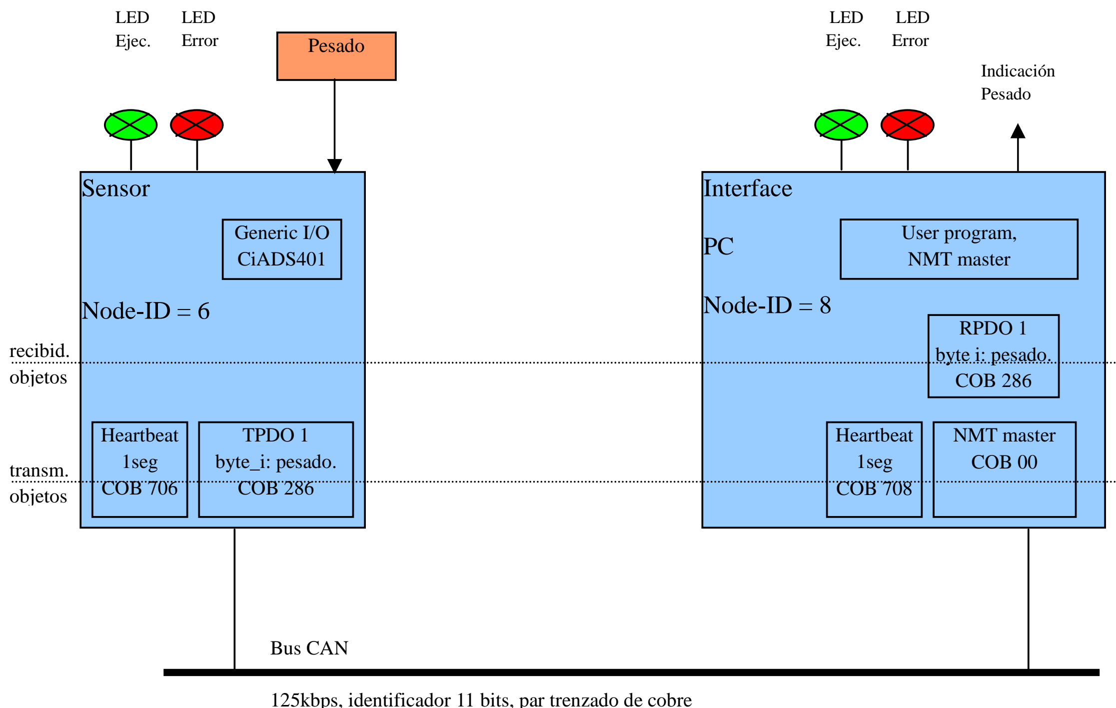


Figura 3. Ejemplo de red CANopen.

2.4 Programa de microcontrolador

Para la elaboración del programa de microcontrolador se requiere conocimientos en el uso de MPLAB IDE y MPLAB C18. Ambos programas deben ser correctamente instalados y configurados.

2.5 Archivos fuente

Se debe descargar el archivo comprimido CANopenNode-v1.10.zip desde el sitio *http://sourceforge.net/projects/canopennode*. Realizamos el 'unzip' del archivo en el disco C:

Luego de la descompresión abrimos el proyecto: Tutorial_Sensor con MPLAB IDE. El proyecto debe ser construido sin errores o advertencias.

Las opciones de folder en *Project>Build options>project>general* son parecidas a la Figura 4. La ruta Include completa, visible en la figura, es:

C:\mcc18\h;C:\CANopenNode_src\CANopen;C:\CANopenNode_src\CANopen\PIC18_with_Microchip_C18;C:\CANopenNode_src\Tutorial_Sensor

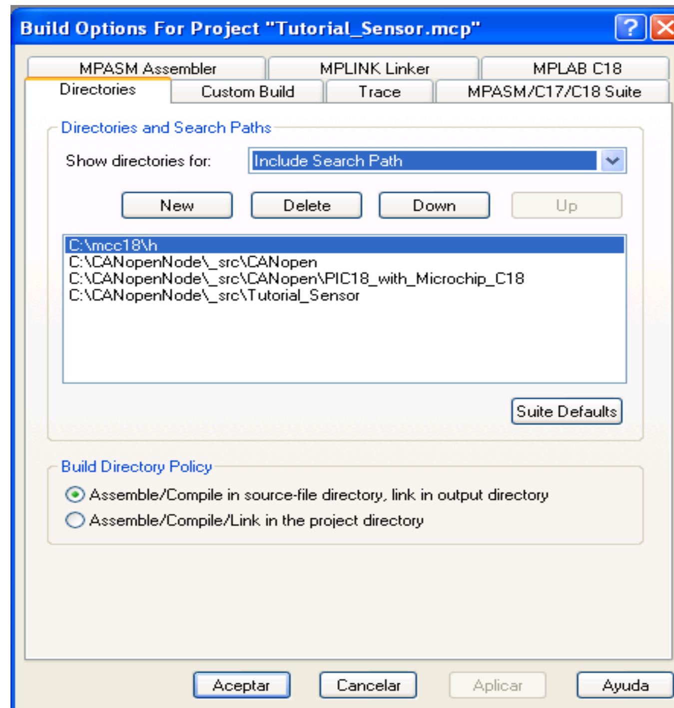


Figura 4. Selección de Folder.

El proyecto contiene los siguientes archivos:

- `src_\lesser.txt` -archivo de licencia
- `src_\CANopen\CANopen.h` -cabecera principal
- `src_\CANopen\CO_errors.h` -definiciones para los errores
- `src_\CANopen\CO_stack.c` -código principal (modificado)
- `src_\CANopen\CO_OD.txt` -descripción del Dicc. de Obj.
- `src_\CANopen\PIC18_with_Microchip_C18\CO_driver.h` -cabecera del driver
- `src_\CANopen\PIC18_with_Microchip_C18\CO_driver.c` -codigo específico de procesador
- `src_\CANopen\PIC18_with_Microchip_C18\main.c` -función main() e interrupciones
- `src_\CANopen\PIC18_with_Microchip_C18\memcpyram2flash.h` -cabecera memoria flash
- `src_\CANopen\PIC18_with_Microchip_C18\memcpyram2flash.c` -escribir a memoria flash
- `src_\CANopen\PIC18_with_Microchip_C18\CONFIG18f458.c` -configuración-opcional
- `src_\CANopen\PIC18_with_Microchip_C18\18f458i.lkr` -script de enlace por defecto
- `src_\Tutorial_xxx\CO_OD.h` -cabecera Diccionario de Objetos
- `src_\Tutorial_xxx\CO_OD.c` -Diccionario de Objetos
- `src_\Tutorial_xxx\user.c` -Código de usuario
- `src_\Tutorial_xxx\Tutorial_xxx.eds` -hoja de datos electrónica

2.6 Programación del sensor

El sensor está basado en el perfil **Ejemplo_Entrada/Salida_genérico** y la especificación CiADS401 – perfil de dispositivos CANopen para módulos de Entrada/Salida genéricos. El sensor, en este caso, la balanza electrónica usará su puerto serial de comunicación RS-232 para la transferencia de lecturas de pesado con el módulo UART incorporado en el PIC y a través de éste, transmitir los datos por la red CAN. (Anexo A para imágenes de conexiones). Se presenta un corto extracto de la especificación CiADS401:

Diccionario de Objetos, índice 0x6000:
Lectura de Entradas Digitales – 64 bits mapeados al TPDO 0 (Transmisión).
Diccionario de Objetos, índice 0x6200:
Escritura de Salidas Digitales – 64 bits mapedos al RPDO 0 (Recepción).
Diccionario de Objetos, índice 0x6400:
Lectura de Entradas Analógicas – 12*16 bit valores mapeados al TPDO 1...3.
Diccionario de Objetos, índice 0x6411:
Escritura de Salidas Analógicas – 12*16 bit valores mapeados al RPDO 1...3.

En nuestra aplicación usamos 16 bytes TPDO2 y TPDO3, mapeados desde el índice 0x6400, subíndice 0x01.

2.7 Instalación del Proyecto – archivo CO_OD.h

Los comentarios en cada línea del código fuente ayudan a comprenderlo mejor.

El código para la sección *Setup CANopen* es:

```
#define CO_NO_SYNC              0
#define CO_NO_EMERGENCY         1
#define CO_NO_RPDO              0
#define CO_NO_TPDO              2
#define CO_NO_SDO_SERVER        1
#define CO_NO_SDO_CLIENT        0
#define CO_NO_CONS_HEARTBEAT    0
#define CO_NO_USR_CAN_RX        0
#define CO_NO_USR_CAN_TX        0
#define CO_MAX_OD_ENTRY_SIZE    20
#define CO_SDO_TIMEOUT_TIME     10
#define CO_NO_ERROR_FIELD       8
#define CO_PDO_PARAM_IN_OD
#define CO_PDO_MAPPING_IN_OD
#define CO_TPDO_INH_EV_TIMER
#define CO_VERIFY_OD_WRITE
#define CO_OD_IS_ORDERED
#define CO_SAVE_EEPROM
#define CO_SAVE_ROM
```

En esta sección se puede definir el número de objetos específico que son usados en CANopenNode. Algunas características pueden ser deshabilitadas.

En nuestro caso el objeto SYNC (sincronía) no será usado, PDO's no serán recibidos, TPDO 0 no será usado (por no tener entradas digitales), TPDO 2 y TPDO3 (16 bytes) serán usados para los valores de pesado desde las balanzas y ningún otro nodo será monitoreado.

El código para la sección *Device profile for Generic I/O* es:

```
    //#define CO_IO_DIGITAL_INPUTS
    //4 * 8 entradas digitales
    //#define CO_IO_DIGITAL_OUTPUTS
//4 * 8 salidas digitales
    #define CO_IO_ANALOG_INPUTS
//16 * 8 bit entradas analógicas
    //#define CO_IO_ANALOG_OUTPUTS
//2 * 16 bit salidas analógicas
```

El código para la sección *Default values for object dictionary* es:

```
#define ODD_PROD_HEARTBEAT   1000
...
#define ODD_ERROR_BEH_COMM   0x01
#define ODD_NMT_STARTUP
0x00000000L
```

El objeto Latido (Heartbeat) se enviará una vez por segundo. Si hay un error de comunicación un bit del Registro de Error (índice 1001) será puesto a uno y el dispositivo estará en el estado NMT Operacional, y permanecerá en ese estado. El comportamiento por defecto es que los errores de comunicación obligan al dispositivo a entrar al estado NMT Pre-Operacional. En el inicio el dispositivo entrará al estado NMT Operacional.

El código para la sección *0x1800 Transmit PDO parameters* es:

```
#define ODD_TPDO_PAR_COB_ID_0 0
#define ODD_TPDO_PAR_T_TYPE_0 255
#define ODD_TPDO_PAR_I_TIME_0 0
#define ODD_TPDO_PAR_E_TIME_0 0
#define ODD_TPDO_PAR_COB_ID_1 0
#define ODD_TPDO_PAR_T_TYPE_1 255
#define ODD_TPDO_PAR_I_TIME_1 1000
#define ODD_TPDO_PAR_E_TIME_1 60000
```

El objeto PDO 0 nunca será enviado (porque la macro CO_IO_DIGITAL_INPUTS esta deshabilitada). El COB-ID (identificador CAN de 11 bits) del PDO 1 será por defecto – 0x280+Node-ID. El tiempo de Inhibición para el PDO 1 es 1000*100µs, de modo que el PDO 1 no será enviado antes de 100 ms. El PDO 1 será enviado en cada cambio de estado y cada minuto (60000 ms) – El tipo de Transmisión es 255 – especificado por el perfil de dispositivo.

El código para la sección *0x1A00 Transmit PDO mapping* para el PDO 1 es:

```
#define ODD_TPDO_MAP_1_1       0x64000110L
#define ODD_TPDO_MAP_1_2       0x64000200L
#define ODD_TPDO_MAP_1_3       0x64000300L
#define ODD_TPDO_MAP_1_4       0x64000400L
#define ODD_TPDO_MAP_1_5       0x00000000L
#define ODD_TPDO_MAP_1_6       0x00000000L
#define ODD_TPDO_MAP_1_7       0x00000000L
#define ODD_TPDO_MAP_1_8       0x00000000L
```

Solo un valor de pesado será enviado en los TPDO's, de modo que sobre el bus CAN este objeto tendrá 16 bytes de datos. La longitud del PDO es calculada automáticamente del mapeo en CANopenNode. Los datos son copiados en cada cambio de estado de la variable *ODE_Read_Analog_Input[num_char]* (localizada en el DO, índice=0x6400, subíndice=1, longitud=0x10 bytes) y donde *num_char* es el número de caracteres del valor de pesado. Para ver cómo trabaja esto, se muestra abajo el código ejemplo del archivo user.c, sección *Write TPDOs*:

```
if(CO_TPDO_InhibitTimer[1] == 0 && (
         CO_TPDO(1).BYTE[i] != ODE_Read_Analog_Input[i])){

     CO_TPDO(1).BYTE[i] = ODE_Read_Analog_Input[i];

     if(ODE_TPDO_Parameter[1].Transmission_type >= 254)
         CO_TPDOsend(1);    }
```

El código para la sección *Default values for user Object Dictionary Entries*:

```
#define ODD_CANnodeID    0x06
#define ODD_CANbitRate   3
```

El Node-ID para el sensor (balanza) es 6 and la tasa de bits CAN es 125 kbps.

2.8 Interface con la aplicación – archivo User.c

El código para la cabecera del archivo:

```
#include <usart.h>
```

Cambie la función *User_Init*:

```
void User_Init(void){
   ...
#ifdef CO_IO_ANALOG_INPUTS
OpenUSART(USART_TX_INT_OFF&USART_RX_INT_ON&USART_ASYNCH_MODE&USART_EIGH
T_BIT&USART_CONT_RX&USART_BRGH_HIGH,25);//B=FOSC/(16*(SPRG+1);9600bps a
//20MHZ

      ODE_Read_Analog_Input[0] = 0;
      ODE_Read_Analog_Input[1] = 0;
      ODE_Read_Analog_Input[2] = 0;
      ODE_Read_Analog_Input[3] = 0;
        ...
      ODE_Read_Analog_Input[12] = 0;
      ODE_Read_Analog_Input[13] = 0;
      ODE_Read_Analog_Input[14] = 0;
      ODE_Read_Analog_Input[15] = 0;
#endif
   ... }
```

Cambie la función *User_Process1msIsr*, sección *Read from Hardware*:

```
//CHANGE THIS LINE -> ODE_Read_Digital_Input.BYTE[0] = port_xxx
//CHANGE THIS LINE -> ODE_Read_Digital_Input.BYTE[1] = port_xxx
//CHANGE THIS LINE -> ODE_Read_Digital_Input.BYTE[2] = port_xxx
//CHANGE THIS LINE -> ODE_Read_Digital_Input.BYTE[3] = port_xxx

if(PIR1bits.RCIF == 1){//Rutina de Interrupción
        ODE_Read_Analog_Input[i] = ReadUSART();
        i++;    }
```

La function es ejecutada cada milisegundo.

El archivo *User.c*, es la conexión entre el código de usuario y la pila CANopenNode. (Anexo B para un resumen de este archivo)

2.9 Compilación del programa y pruebas

Compilamos el proyecto y programamos el PIC. Reconectamos el PIC a la fuente de energía para habilitarlo. Sin conectar aún a la red, después del encendido el LED verde de ejecución CAN brilla. Esto significa que el estado NMT es el Operacional. LED rojo de error CAN brilla una vez repetidamente. Esto significa que el bus CAN está en estado pasivo. Si hacemos un 'corto circuito´ sobre las patillas de las señales CAN_LO y CAN_HI el LED rojo de error CAN encenderá. Esto significa que el bus CAN esta Off. El Sensor no dejará el estado Operacional a causa de los errores de comunicación. En nuestro caso eso es correcto.

3. Resultados

Utilizando el adaptador de PC USB a bus CAN y el software de monitoreo de bus CAN instalado en el nodo maestro (Anexo A para imágenes del adaptador y pantallas de mensajes CAN recibidos), podemos observar los COB-ID's correspondientes a TPDO 1 y TPDO2, siendo los bytes de datos de pesado embebidos, transmitidos por el nodo sensor.

Se desarrolló la aplicación de usuario para el Sistema Operativo Windows® en Delphi® XE6 (Figura 5) que permite la visualización en formato ASCII del valor de pesado, la aplicación también se encarga de guardar la información de registro en un archivo XML. En la Figura 6 se puede observar los datos de pesado recibidos en formato ASCII y decimal.

Los mensajes transferidos desde la balanza electrónica a través de la red CANOpen tienen en realidad dos identificadores 286h y 386h (representación hexadecimal) que son visualizados con la ayuda de un programa monitor de mensajes CAN (Pcan View de PEAK® System). Los dígitos en la lectura de la balanza son representados por caracteres en código ASCII en el programa monitor. Ejemplo: 30 en código ASCII, dígito 0; 31 en código ASCII, dígito 1; 2E en código ASCII, caracter punto; 67 en código ASCII, caracter g; 00, 20, 0D en código ASCII, son caracteres no imprimibles.

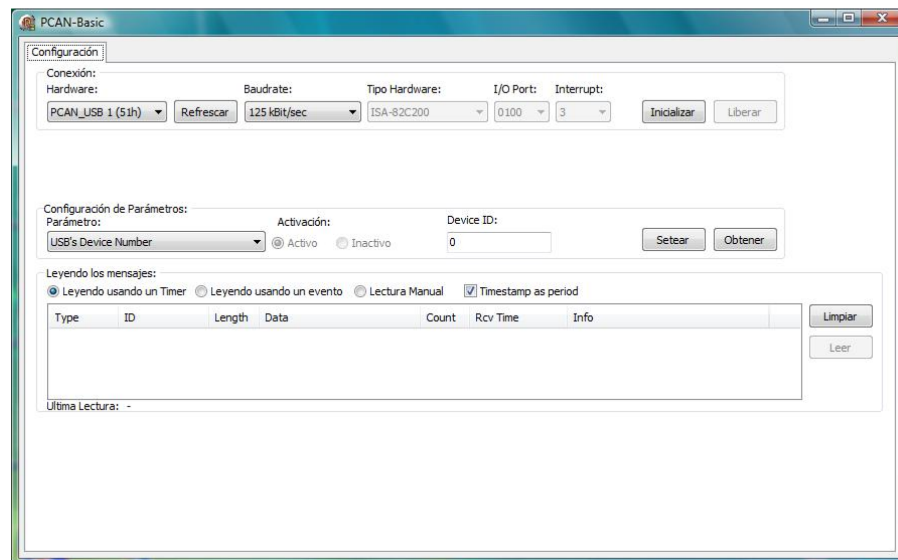


Figura 5. Interface de programa de usuario.

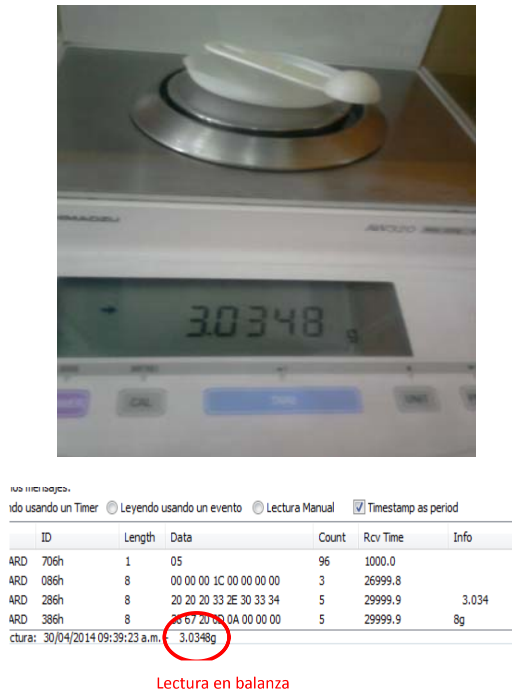


Figura 6. Lectura y registro de valor de pesado en programa de usuario.

4. Agradecimientos

A la División de Técnicas Analíticas Nucleares de la Dirección de Investigación y Desarrollo del IPEN por todas las facilidades brindadas para el desarrollo e instalación del estándar. Al OIEA a través del ARCAL CXXIII: "Apoyo a la automatización de sistemas y procesos en instalaciones nucleares", cuyo coordinador es el Ing. Bruno Mendoza y a los Ing. Ever Cifuentes y Boris Ninapaytan por su aporte y discusión al presente trabajo.

5. Bibliografía

[1]. Pfeiffer O, Ayre A, Keydel C. Embendded networking with CAN and CANopen. RTC Books; 2003.
[2]. Baltuano O, Bedregal P, Montoya E. Propuesta técnica para la implementación de una red integrada de control y comunicaciones de instrumentos analíticos (RICCIA). Instituto Peruano de Energía Nuclear. Dirección de Investigación y Desarrollo. 2010. [Informe Interno].
[3]. Robert Bosch GmbH. CAN Specification 2.0. 1991.
[4]. GNU Lesser General Public License, Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
[5]. Paternoster J. CANopenNode Manual v.1.10. December 2005. Disponible en: http://www.siegels.us/HomeAutomation/CANOpen_Node_SGS/Manual.pdf
[6]. CiA. CANopen Application Layer and Communication Profile. [serie en Internet]. Disponible en: http://www.can-cia.org/
[7]. CiA. CANopen Device Profile for Generic I/O Modules. CiADS401, Draft Standard 401, Version Version 2.1.
[8]. CiA. CANopen Indicator Specification. CiADR303-3: Draft Recommendation 303-3, Version 1.0.

Anexo A

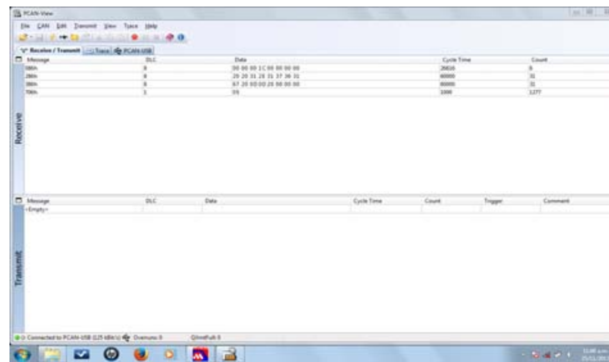
Fotografías de la instalación implementada y pantallas de resultados

Fotografía 1. Puerto Serial RS-232 de la balanza (Sensor)

Fotografía 2. Conexión de la balanza al UART del PIC 18F458

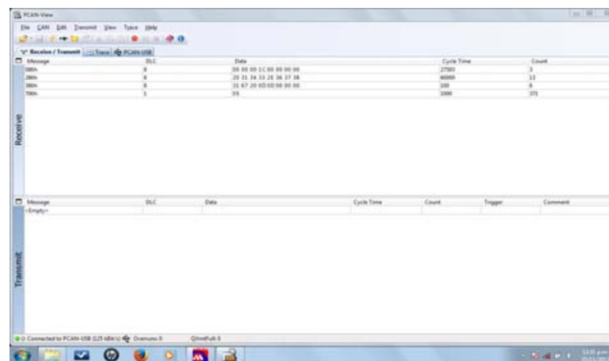
Fotografía 3. Adaptador de PC USB a bus CAN

Fotografía 4. Lectura en balanza mostrando un digito entero y cuatro decimales.



Fotografía 5. Mensaje CANopen correspondiente a lectura anterior

Fotografía 6. Lectura en balanza con tres dígitos enteros y cuatro decimales



Fotografía 7. Mensaje CANopen correspondiente a la última lectura

Anexo B

Listado de programa en lenguaje C de función *main.c* de la aplicación firmware:

```
/*******************************************************************************
   main.c - Main and ISR functions - Processor specific
               Processor: PIC18Fxxx
               Compiler: MPLAB - C18 (Microchip) >= V3.00

   Author: janez.paternoster@siol.net
*******************************************************************************/
#include "CANopen.h"
/* Function prototypes *******************************************************/
   //from CO_driver.c
   void CO_CANrxIsr(void);       //ISR for receiving CAN messages
   void CO_CANtxIsr(void);       //ISR for transmiting CAN messages
   void CO_DriverInit(void);
   //from CO_stack.c
   void CO_ResetComm(void);      //Reset communication
   void CO_Timer1msIsr(void);    //Process CANopen from 1ms timer interrupt,sync.
   void CO_ProcessMain(void);    //Process CANopen from main function, async.
   //from user.c
   void User_Init(void);
   void User_ProcessMain(void);
   #ifdef USER_ISR_HIGH
      void User_IsrHigh(void);
   #endif
   #ifdef USER_ISR_LOW
      void User_IsrLow(void);
   #endif

/* Parameters for Timer2 (1ms interrupt) *************************************/
   #if CO_OSCILATOR_FREQ == 20
      #define TIMER2PRESCALE     1     //0=1:1, 1=1:4, 2=1:16
      #define TIMER2POSTSCALE    10    //1...16
      #define TIMER2PR2          124   //0...255
   #else
      #error define_CO_OSCILATOR_FREQ CO_OSCILATOR_FREQ not supported
   #endif

/* INTERRUPTS ****************************************************************/
#pragma interruptlow IsrLow
void IsrLow (void){
   if(PIR1bits.TMR2IF){
      PIR1bits.TMR2IF = 0;
      CO_Timer1msIsr();
      if(PIR1bits.TMR2IF) ErrorReport(ERROR_isr_timer_overflow, 0);
      }
  #ifdef USER_ISR_LOW
   User_IsrLow();
  #else
   else
      ErrorReport(ERROR_isr_low_WrongInterrupt, 0);
  #endif
}

#pragma code high_vector=0x08
void interrupt_high(void){
  #ifdef USER_ISR_HIGH
   _asm
```

```
      MOVF PIR3, 0, ACCESS
      ANDLW 0x3
      BZ M_LAB_1
      GOTO CO_CANrxIsr
     M_LAB_1:
      GOTO User_IsrHigh
   _endasm
  #else
   _asm
      GOTO CO_CANrxIsr
   _endasm
  #endif
}
#pragma code

#pragma code low_vector=0x18
void interrupt_low(void){
   _asm
      BTFSC PIR3, 0x2, ACCESS   //test TXB0 interrupt flag, skip line if cleared
      GOTO CO_CANtxIsr
      GOTO IsrLow
   _endasm
}
#pragma code

/******* main() *******/
void main (void){
   CO_DriverInit();  //CANopenNode driver init
   User_Init();      //User init
   CO_ResetComm();   //Reset communication

   T2CON = ((TIMER2POSTSCALE-1) << 3) | 0x04 | TIMER2PRESCALE;
   PR2 = TIMER2PR2;
   TMR2 = 0;
   //enable timer interrupts
   PIR1bits.TMR2IF = 0;
   IPR1bits.TMR2IP = 0;   //low priority interrupt
   PIE1bits.TMR2IE = 1;
   //interrupts
   RCONbits.IPEN   = 1;    //interrupt priority enable
   INTCONbits.GIEL = 1;    //global interrupt low enable
   //INTCONbits.GIEH = 1;    //global interrupt high enable
   INTCONbits.GIE = 1;      //global interrupt enable

   while(1){
      ClrWdt();
      CO_ProcessMain();
      ClrWdt();
      User_ProcessMain();
   }
}
```

Listado en lenguaje C del archivo *User.c* (Listado parcial) de la aplicación firmware

```
#include "CANopen.h"
#include <usart.h>

/***** Device profile for Generic I/O ****************************************/
           #ifdef CO_IO_DIGITAL_INPUTS
/*0x6000*/ extern       tData4bytes    ODE_Read_Digital_Input;
           #endif

           #ifdef CO_IO_DIGITAL_OUTPUTS
/*0x6200*/ extern       tData4bytes    ODE_Write_Digital_Output;
           #endif

           #ifdef CO_IO_ANALOG_INPUTS
/*0x6400*/ extern       INTEGER8      ODE_Read_Analog_Input[];
           #endif

           #ifdef CO_IO_ANALOG_OUTPUTS
/*0x6411*/ extern       INTEGER16      ODE_Write_Analog_Output[];
           #endif
. . .

/*******************************************************************************
   User_Init - USER INITIALIZATION OF NODE
   Function is called after start of program.
*******************************************************************************/
void User_Init(void){
   ODE_EEPROM.PowerOnCounter++;

   #ifdef CO_IO_DIGITAL_INPUTS
      //CHANGE THIS LINE -> set ports as digital inputs
      ODE_Read_Digital_Input.DWORD[0] = 0;
   #endif

   #ifdef CO_IO_ANALOG_INPUTS
      TRISC = 0b10000000;

OpenUSART(USART_TX_INT_OFF&USART_RX_INT_ON&USART_ASYNCH_MODE&USART_EIGHT_BIT&USAR
T_CONT_RX&USART_BRGH_LOW,129); //B=FOSC/(64*(SPRG+1);//2400BPS

      ODE_Read_Analog_Input[0] = 0;
      ODE_Read_Analog_Input[1] = 0;
      ODE_Read_Analog_Input[2] = 0;
      ODE_Read_Analog_Input[3] = 0;
      ODE_Read_Analog_Input[4] = 0;
      ODE_Read_Analog_Input[5] = 0;
      ODE_Read_Analog_Input[6] = 0;
      ODE_Read_Analog_Input[7] = 0;
      ODE_Read_Analog_Input[8] = 0;
      ODE_Read_Analog_Input[9] = 0;
      ODE_Read_Analog_Input[10] = 0;
      ODE_Read_Analog_Input[11] = 0;
      ODE_Read_Analog_Input[12] = 0;
      ODE_Read_Analog_Input[13] = 0;
      ODE_Read_Analog_Input[14] = 0;
      ODE_Read_Analog_Input[15] = 0;
   #endif
. . .


/*******************************************************************************
   User_ProcessMain - USER PROCESS MAINLINE
   This function is cyclycally called from main(). It is non blocking function.
   It is asynchronous. Here is longer and time consuming code. Cycle time can
```

```
   be shorter than in User_Process1msIsr.
*****************************************************************************/
volatile char count;
void User_ProcessMain(void){
       if (count == 13)
              count = 0;
}

void User_Process1msIsr(void){
   static unsigned char LastStateOperationalGradePrev = 0;
   unsigned char LastStateOperationalGrade = 0;
   extern volatile unsigned int CO_TPDO_InhibitTimer[CO_NO_TPDO];  //Inhibit
timer used //for inhibit PDO sending - if 0, TPDO is not inhibited

   //PDO Communication
   if(CO_NMToperatingState == NMT_OPERATIONAL){

      LastStateOperationalGrade ++;
. . .
      //verify operating state of monitored nodes
      #if CO_NO_CONS_HEARTBEAT > 0
      if(CO_HBcons_AllMonitoredOperational == NMT_OPERATIONAL){
      #endif

         LastStateOperationalGrade ++;

      #if CO_NO_CONS_HEARTBEAT > 0
      }// end if(CO_HBcons_AllMonitoredOperational == NMT_OPERATIONAL)
      #endif


      //Write TPDOs -----------------------------------------------------------
      //Following code realizes Static TPDO Mapping
      //Transmission is Synchronous or Change of State, depends on
Transmission_type.
      //Inhibit timer and Periodic Event Timer can be used.
      #if CO_NO_TPDO > 0
         #ifdef CO_IO_DIGITAL_INPUTS
            if(CO_TPDO_InhibitTimer[0] == 0 &&
               CO_TPDO(0).DWORD[0] != ODE_Read_Digital_Input.DWORD[0]){

               CO_TPDO(0).DWORD[0] = ODE_Read_Digital_Input.DWORD[0];
               if(ODE_TPDO_Parameter[0].Transmission_type >= 254)
                  CO_TPDOsend(0);
            }
         #endif
      #endif
      #if CO_NO_TPDO > 1
         #ifdef CO_IO_ANALOG_INPUTS
            if((CO_TPDO_InhibitTimer[1] == 0 &&
               CO_TPDO(1).BYTE[0] != ODE_Read_Analog_Input[0] ||
               CO_TPDO(1).BYTE[1] != ODE_Read_Analog_Input[1] ||
               CO_TPDO(1).BYTE[2] != ODE_Read_Analog_Input[2] ||
               CO_TPDO(1).BYTE[3] != ODE_Read_Analog_Input[3] ||
               CO_TPDO(1).BYTE[4] != ODE_Read_Analog_Input[4] ||
               CO_TPDO(1).BYTE[5] != ODE_Read_Analog_Input[5] ||
               CO_TPDO(1).BYTE[6] != ODE_Read_Analog_Input[6] ||
               CO_TPDO(1).BYTE[7] != ODE_Read_Analog_Input[7])){
               CO_TPDO(1).BYTE[0] = ODE_Read_Analog_Input[0];
               CO_TPDO(1).BYTE[1] = ODE_Read_Analog_Input[1];
               CO_TPDO(1).BYTE[2] = ODE_Read_Analog_Input[2];
               CO_TPDO(1).BYTE[3] = ODE_Read_Analog_Input[3];
               CO_TPDO(1).BYTE[4] = ODE_Read_Analog_Input[4];
               CO_TPDO(1).BYTE[5] = ODE_Read_Analog_Input[5];
               CO_TPDO(1).BYTE[6] = ODE_Read_Analog_Input[6];
               CO_TPDO(1).BYTE[7] = ODE_Read_Analog_Input[7];
               if(ODE_TPDO_Parameter[1].Transmission_type >= 254)
                  CO_TPDOsend(1);
```

```
            }
         #endif
      #endif
      #if CO_NO_TPDO > 2
         #ifdef CO_IO_ANALOG_INPUTS
            if((CO_TPDO_InhibitTimer[2] == 0 &&
               CO_TPDO(2).BYTE[0] != ODE_Read_Analog_Input[8] ||
               CO_TPDO(2).BYTE[1] != ODE_Read_Analog_Input[9] ||
               CO_TPDO(2).BYTE[2] != ODE_Read_Analog_Input[10] ||
               CO_TPDO(2).BYTE[3] != ODE_Read_Analog_Input[11] ||
               CO_TPDO(2).BYTE[4] != ODE_Read_Analog_Input[12] ||
               CO_TPDO(2).BYTE[5] != ODE_Read_Analog_Input[13] ||
               CO_TPDO(2).BYTE[6] != ODE_Read_Analog_Input[14] ||
               CO_TPDO(2).BYTE[7] != ODE_Read_Analog_Input[15])){
               CO_TPDO(2).BYTE[0] = ODE_Read_Analog_Input[8];
               CO_TPDO(2).BYTE[1] = ODE_Read_Analog_Input[9];
               CO_TPDO(2).BYTE[2] = ODE_Read_Analog_Input[10];
               CO_TPDO(2).BYTE[3] = ODE_Read_Analog_Input[11];
               CO_TPDO(2).BYTE[4] = ODE_Read_Analog_Input[12];
               CO_TPDO(2).BYTE[5] = ODE_Read_Analog_Input[13];
               CO_TPDO(2).BYTE[6] = ODE_Read_Analog_Input[14];
               CO_TPDO(2).BYTE[7] = ODE_Read_Analog_Input[15];
               if(ODE_TPDO_Parameter[2].Transmission_type >= 254)
                  CO_TPDOsend(2);
            }
         #endif
      #endif


   } //end if(CO_NMToperatingState == NMT_OPERATIONAL)


   if(LastStateOperationalGrade < LastStateOperationalGradePrev){
//NMT_OPERATIONAL (this or monitored nodes) was just lost
      SwitchOffNode();
   }
   LastStateOperationalGradePrev = LastStateOperationalGrade;
}

#pragma interrupt User_IsrSerial
void User_IsrSerial(void){
   if(PIR1bits.RCIF == 1){//Running ISR
   //*************processing interrupt source**********//
      ODE_Read_Analog_Input[count] = ReadUSART();
              count++;
   }
}
```