Cégep Limoilou

Électronique Programmable et Robotique

247-6[1-2-3-4]7-LI

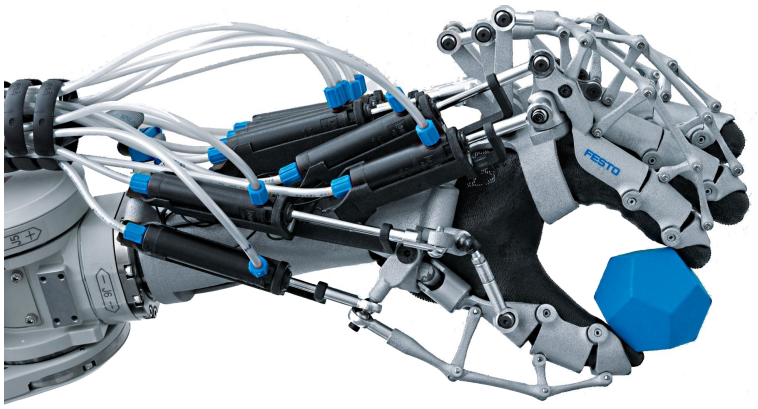
Projet de 5e session

Étudiants :

Vincent Chouinard
Hicham Safoine
Gabriel Fortin-Bélanger
Louis-Nomand Ang-Houle

Professeurs :

Ali Tadli
Alain Champagne
Stéphane Deschênes
Étienne Tremblay



L'usine à gaz, et le gaz, c'est de l'air !

20 novembre 2014

Table des matières

Table des figures

Liste des tableaux

1 Présentation du projet

Le projet de la cinquième session consiste à réaliser

⇒ Le Bolide

⇒ Carte Dallas DS89C450

⇒ Carte uPSD 3254A

⇒ SOC8200

⇒ Table FESTO

⇒ Carte PIC16F88

⇒ Carte d'extension I2C

⇒ Carte d'extension SPI

⇒ Une pile de 10.8 volts

⇒ Quatre moteurs et autant de pneus

1.1 Explication du projet

1.2 Schéma bloc du système

1.2.1 Bloc 1

Le bloc 1 est composé d'un ordinateur muni d'une carte d'extension PCI vers bus CAN. Son rôle est de contrôler et diriger toute l'opération et de veiller au bon fonctionnement de chaque composantes à l'aide d'une application en Csharp. Le bloc 1 est le cerveau de l'usine.

1.2.2 Bloc 2

Le bloc 2 est composé d'un système embarqué Linux basé sur le SOC8200. Son rôle principal est d'agir comme sniffeur d'information et d'afficher sur son écran toutes les données qui transitent sur le bus CAN. Toutefois, ce dernier est en mesure de détecter une défaillance du PC via un gestionnaire de HeartBeat et de prendre la relève en tant que cerveau de l'opération. Le SOC8200 agit comme vice-président du bus CAN.

1.2.3 Bloc 3

1.2.4 Bloc 4

1.3 Liste des logiciels

Terminaux

- UART Master 1.0.3
- Serializ3r 1.0.2
- TerraTerm
- Putty
- GTKterm 0.99.7-rc1
- xTerminator
- CAPS
- tinyBootloader

Gestion du projet

- MS Project 2012
- Git Hub

Compilateurs et IDE

- Visual Studio 2013
- Visual Studio 2010
- IAR 8.20
- MPLAB

Éditeur de texte

- Notepad++
- gedit
- medit 1.2.0

Schémas électriques

- OrCAD 16.2

Système d'exploitation

- Windows 7 SP1
- Windows 8.1
- Windows XP SP3
- Fedora 20
- CentOS
- Lubuntu 14.10

Autres

- VMWare Workstation 10
- TeXmaker 4.3
- Dukto R6
- Dia
- Festo configuration tool

1.4 Liste des trames

TABLE 1 – Index des identifiants matériel CAN

Device	ID matériel
Ordinateur	000
SOC8200	001
Station 1	002
Station 2	003
Station 3	004
Véhicule	005

TABLE 2 – Index des trames CAN

Fonctionnalité	Composante	Données	TimeStamp
Démarre le véhcule	0x00	0x00	TimeStamp
Arrête le véhicule	0x00	0x01	TimeStamp
Le véhicule est arrêté	0x01	0x00	TimeStamp
Le véhicule est en marche	0x01	0x01	TimeStamp
Le véhicule est hors circuit	0x01	0x02	TimeStamp
Vitesse (0-100)	0x02	0x00 à 0x64	TimeStamp
Battrie	0x03	0x00 à 0x64	TimeStamp
Couleur du bloc	0x04	0x00 à 0x02	TimeStamp
Poids du bloc	0x05	0x00 à 0x64	TimeStamp
Envoyer l'heure	0x06	à déterminer	TimeStamp
No. de la station	0x07	0x00 à 0x02	TimeStamp
Demande de l'historique	0xC0	0x00	TimeStamp
Direction horaire et antihoraire	0x08	0x00 à 0x01	TimeStamp

TABLE 3 – Index des communications CAN

Émetteur	Action	ID receveur	Donnée envoyée	TimeStamp	Récepteur	Erreur
Ordinateur	Démarrer le véhicule	004	00 00	TimeStamp	Véhicule	F1
Ordinateur	Arrêter le véhicule	004	00 01	TimeStamp	Véhicule	F2
Véhicule	Dit : je suis arrêté	000	01 00	TimeStamp	Ordinateur	F3
Véhicule	Dit : j'avance	000	01 01	TimeStamp	Ordinateur	F4
Véhicule	Dit : je suis hors circuit	000	01 02	TimeStamp	Ordinateur	F5
Véhicule	Dit sa vitesse	000	02 [00 à 64]	TimeStamp	Ordinateur	F6
Véhicule	Dit le niveau de sa battrie	000	03 [00 à 64]	TimeStamp	Ordinateur	F7
Station 1	Dit bloc = métal	000	04 00	TimeStamp	Ordinateur	F8
Station 1	Dit bloc = orange	000	04 01	TimeStamp	Ordinateur	F9
Station 1	Dit bloc = noir	000	04 02	TimeStamp	Ordinateur	FA
Station 1	Dit le poid du bloc	000	05 [00 à 64]	TimeStamp	Ordinateur	FB
Voiture	Dit qu'elle est à la station 1	000	07 00	TimeStamp	Ordinateur	FC
Voiture	Dit qu'elle est à la station 2	000	07 01	TimeStamp	Ordinateur	FD
Ordinateur	Envoie l'heure	003	06 à déterminer	TimeStamp	Station 1	FE
Ordinateur	Demande le LOG	001	C0 00	TimeStamp	SOC8200	E0
Ordinateur	Exige Horaire	004	08 00	TimeStamp	Véhicule	E1
Ordinateur	Exige Antihoraire	004	08 01	TimeStamp	Véhicule	E2

Note : Il faut définir les TimeStamps et la checkSUM

Note : La station no.1 relaie les données entre l'ordinateur et la station no.3 (pesage), entre l'ordinateur et la station no.2 (table Festo) via xbee et entre la voiture et le PC via Xbee.

Note : FF, c'est la checkSUM, mais elle n'a pas encore été faite

Note : Il faut ajouter le TimeStamp

```
CAN.SendToPC("0100FF"); // Arrêté
CAN.SendToPC("0101FF"); // En marche
CAN.SendToPC("0102FF"); // Hors circuit
CAN.SendToPC("02xxFF"); // Vitesse de xx
CAN.SendToPC("03xxFF"); // Battrie chargée à xx %
CAN.SendToPC("0400FF"); // Bloc métallique
CAN.SendToPC("0401FF"); // Bloc noire
CAN.SendToPC("0402FF"); // Bloc orange
CAN.SendToPC("050064"); // Le bloc est lourd
CAN.SendToPC("0700FF"); // Rendu à la station 1
CAN.SendToPC("0701FF"); // Rendu à la station 2
CAN.SendToPC("0702FF"); // Rendu à la station 3
```

f

2 Le matériel

2.1 Bloc 1

D'un pont de vu matériel, le bloc 1 est le plus simple, car il est composé d'un ordinateur Windows munie d'une carte PCI vers CAN et d'une prise RS232. Il n'y a rien à faire, mise à part brancher les bons câbles aux bons endroits.

2.2 Bloc 2

2.3 Bloc 3

2.4 Bloc 4

2.5 Explication des types de liens

2.5.1 RS232

Un lien RS232 9600 Bauds est établi entre l'ordinateur et le SOC8200. Ce lien sert à l'envoie et à la réception de HeartBeat, afin que le SOC8200 ou l'ordinateur soit informé de toute défaillance de l'autre.

2.5.2 Xbee

Lorsque les modules Xbee sont adéquatement configurés, ils font office de remplacement au câble RS232. En effet, nos Xbee discutent entre eux à l'aide du protocole de communication RS232 à 9600 bauds.

2.6 Explication des trames

2.6.1 RS-232

Le protocole RS-232 sert à envoyer et à recevoir des HeartBeat. Le PC et le SOC 8200 s'envoient tous deux un HeartBeat par seconde à 9600 bauds. Un HeartBeat, c'est simplement le mot "Allo". Le PC et le SOC2800 "écoutent" les HeartBeats, et si ces derniers ne sont pas entendus, chaque dispositif prend pour acquis que l'autre est hors-service et prend la relève de la gestion du bus CAN.

2.6.2 CAN

2.6.3 XBEE

Trois modules Xbee sont présent sur l'ensemble du projet, soit sur la station no.1 (la carte μPSD), sur la station no.2 (la table FESTO) et la station no.6, c'est à dire le bolide. La particularité des Xbee est que lorsqu'ils sont adéquatement configurés, tout ce qu'envoie un Xbee est reçu et lu par tous les autres Xbee à proximité, et c'est pourquoi nous avons définit un système de trames.

2.7 Liste des pièces

- Carte Dallas
- Carte uPSD
- SOC 8200
- PIC18Fmachin
- Carte d'extension SPI
- Carte d'extension I2C
- Carte CAN MCP2515
- XBEE
- Table FESTO
- Carte d'extension IO
- Carte connecteur DAC ADC
- Carte Xbee vers DB9
-
-
-
-
-
-
-
-
-
-
-

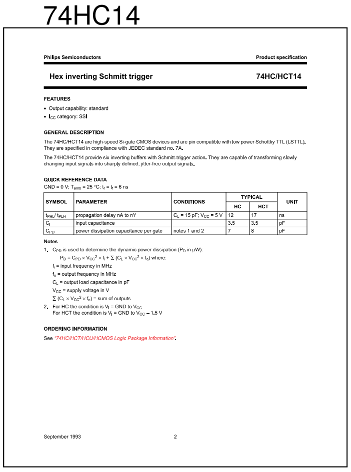
2.7.1 Liens web

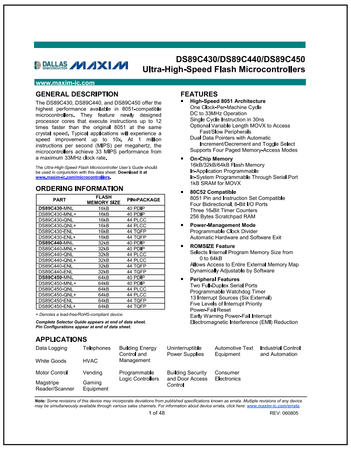
Mettre ien vers GitHUB

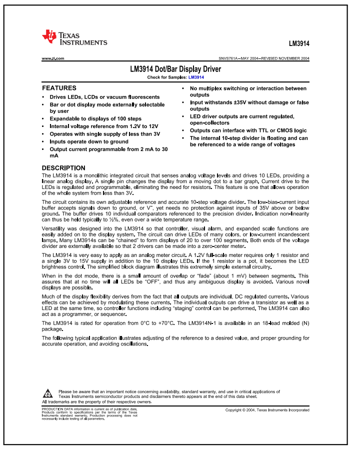
2.7.2 Datasheets

Note : Toutes les datasheets sont en format non-compressé. Vous pouvez zoomer sur le document PDF [1] afin de lire l'intégralité de leurs première page.

74HC14

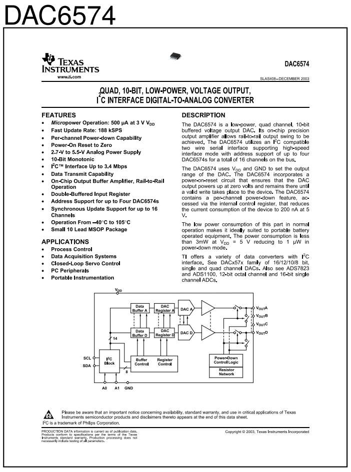

Philips Semiconductors Product specification
Hex inverting Schmitt trigger 74HC/HCT14


DS89C430/DS89C440/DS89C450
Ultra-High-Speed Flash Microcontrollers

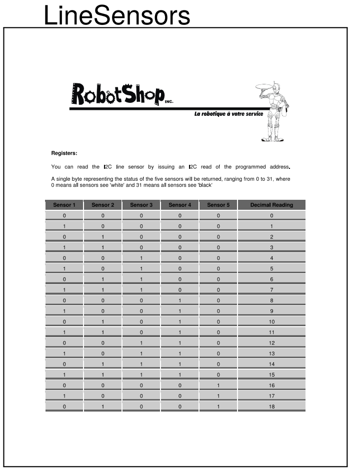

LM3914
LM3914 Dot/Bar Display Driver

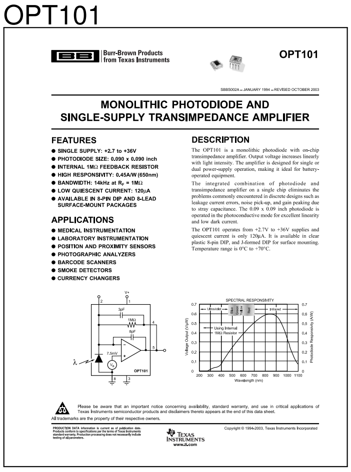
OPT101

DAC6574


DAC6574
QUAD, 10-BIT, LOW-POWER, VOLTAGE OUTPUT, I²C INTERFACE DIGITAL-TO-ANALOG CONVERTER

LineSensors


RobotShop

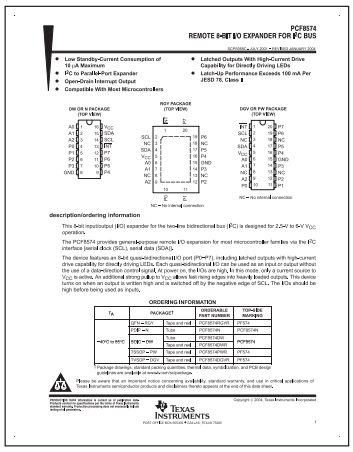

OPT101
MONOLITHIC PHOTODIODE AND SINGLE-SUPPLY TRANSIMPEDANCE AMPLIFIER

DS89C450

LM3914

PCF8573

1. Présenter les datasheets de la sorte économise du papier, donc des arbres, mais requiert de consulter le document .PDF afin de lire adéquatement les datasheets.



PCF8574
REMOTE 8-BIT I/O EXPANDER FOR I²C BUS

- Low Standby-Current Consumption of 10 µA Maximum
- I²C to Parallel-Port Expander
- Open-Drain Interrupt Output
- Compatible With Most Microcontrollers
- Latched Outputs With High-Current Drive Capability for Directly Driving LEDs
- Latch-Up Performance Exceeds 100 mA Per JESD 78, Class II

description/ordering information

This 8-bit input/output (I/O) expander for the two-line bidirectional bus (I²C) is designed for 2.5-V to 6-V V_{CC} operation.

The PCF8574 provides general-purpose remote I/O expansion for most microcontroller families via the I²C interface [serial clock (SCL), serial data (SDA)].

The device features an 8-bit quasi-bidirectional I/O port (P0–P7), including latched outputs with high-current drive capability for directly driving LEDs. Each quasi-bidirectional I/O can be used as an input or output without the use of a data-direction control signal. At power on, the I/Os are high. In this mode, only a current source to V_{CC} is active. An additional strong pullup to V_{CC} allows fast rising edges into heavily loaded outputs. This device turns on when an output is written high and is switched off by the negative edge of SCL. The I/Os should be high before being used as inputs.

ORDERING INFORMATION

TA	PACKAGE†		ORDERABLE PART NUMBER	TOP-SIDE MARKING
−40°C to 85°C	QFN – RGY	Tape and reel	PCF8574RGYR	PF574
	PDIP – N	Tube	PCF8574N	PCF8574N
	SOIC – DW	Tube	PCF8574DW	PCF8574
		Tape and reel	PCF8574DWR	
	TSSOP – PW	Tape and reel	PCF8574PWR	PF574
	TVSOP – DGV	Tape and reel	PCF8574DGVR	PF574

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this data sheet.

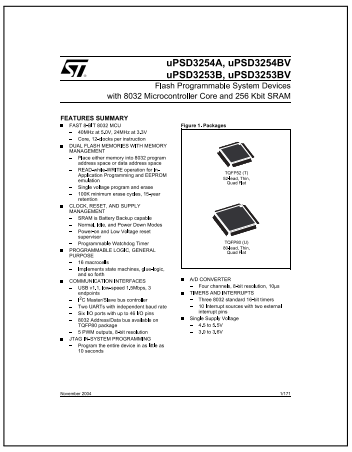
PRODUCTION DATA information is current as of publication date. Products conform to specifications per the terms of Texas Instruments standard warranty. Production processing does not necessarily include testing of all parameters.

Copyright © 2004, Texas Instruments Incorporated

TEXAS INSTRUMENTS
POST OFFICE BOX 655303 • DALLAS, TEXAS 75265

1

uPSD3254A



uPSD3254A, uPSD3254BV
uPSD3253B, uPSD3253BV
Flash Programmable System Devices
with 8032 Microcontroller Core and 256 Kbit SRAM

FEATURES SUMMARY

- FAST 8-BIT 8032 MCU
  - 40MHz at 5.2V, 24MHz at 3.2V
  - 4 clocks per 8032 cycle, 1 per instruction
- DUAL FLASH MEMORIES WITH MEMORY MANAGEMENT
  - Place either memory into 8032 program address space or data address space
  - READ-while-WRITE operation for In-Application Programming and EEPROM emulation
  - Single-voltage program and erase
  - 100K minimum erase cycles, 15-year retention
- CLOCK, RESET, AND SUPPLY MANAGEMENT
  - SRAM is Battery Backup capable
  - Normal, Idle, and Power Down Modes
  - Powerup and Low Voltage reset circuitry
  - Programmable Watchdog Timer
- PROGRAMMABLE LOGIC, GENERAL PURPOSE
  - 16 macrocells
  - Implements state machines, glue logic, and so forth
- COMMUNICATION INTERFACES
  - USB v1.1, low-speed 1.5Mbps, 3 endpoints
  - I²C Master/Slave bus controller
  - Two UARTs with independent baud rate
  - Six I/O ports with up to 46 I/O pins
  - 8032 Address/Data bus available on TQFP80 package
  - 5 PWM outputs, 8-bit resolution
- JTAG In-SYSTEM PROGRAMMING
  - Program the entire device in as little as 15 seconds

Figure 1. Packages

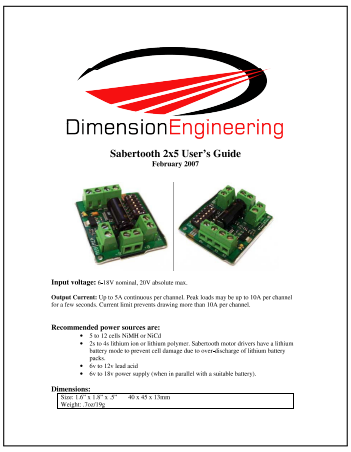
TQFP52 (T)
52-lead, Thin, Quad Flat

TQFP80 (U)
80-lead, Thin, Quad Flat

- A/D CONVERTER
  - Four channels, 8-bit resolution, 10µs
- TIMERS AND INTERRUPTS
  - Three 8032 standard 16-bit timers
  - 10 interrupt sources with two external interrupt pins
- Single-Supply Voltage
  - 4.5 to 5.5V
  - 3.0 to 3.6V

November 2004 1/171

SaberTooth motor drive



DimensionEngineering

Sabertooth 2x5 User's Guide
February 2007

Input voltage: 6-18V nominal, 20V absolute max.

Output Current: Up to 5A continuous per channel. Peak loads may be up to 10A per channel for a few seconds. Current limit prevents drawing more than 10A per channel.

Recommended power sources are:

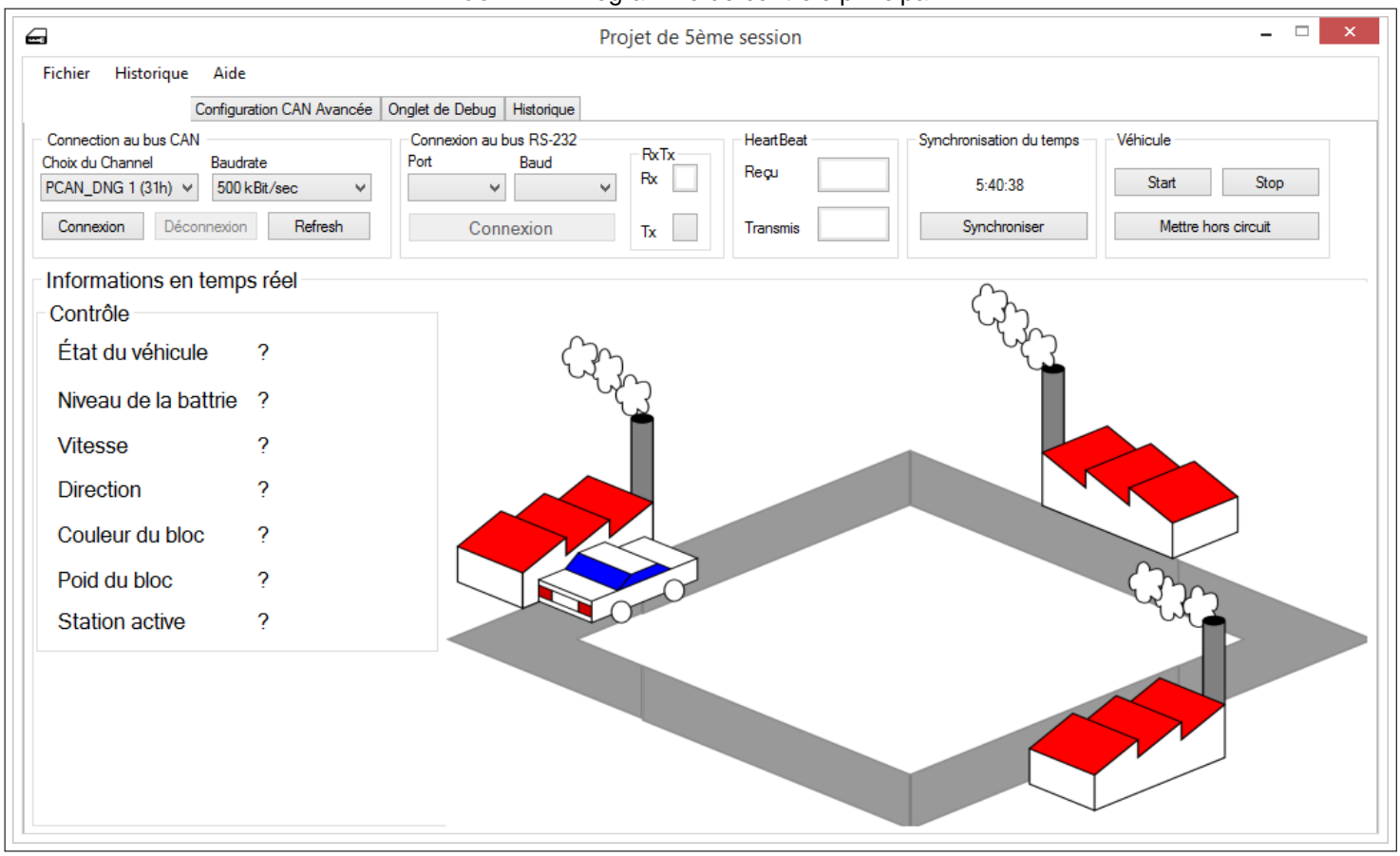
- 5 to 12 cells NiMH or NiCd
- 2s to 4s lithium ion or lithium polymer. Sabertooth motor drivers have a lithium battery mode to prevent cell damage due to over-discharge of lithium battery packs.
- 6v to 12v lead acid
- 6v to 18v power supply (when in parallel with a suitable battery).

Dimensions:

Size: 1.6" x 1.8" x .5"	40 x 45 x 13mm
Weight: .7oz/19g	

3 Interface PC

FIGURE 1 – Programme de contrôle principal

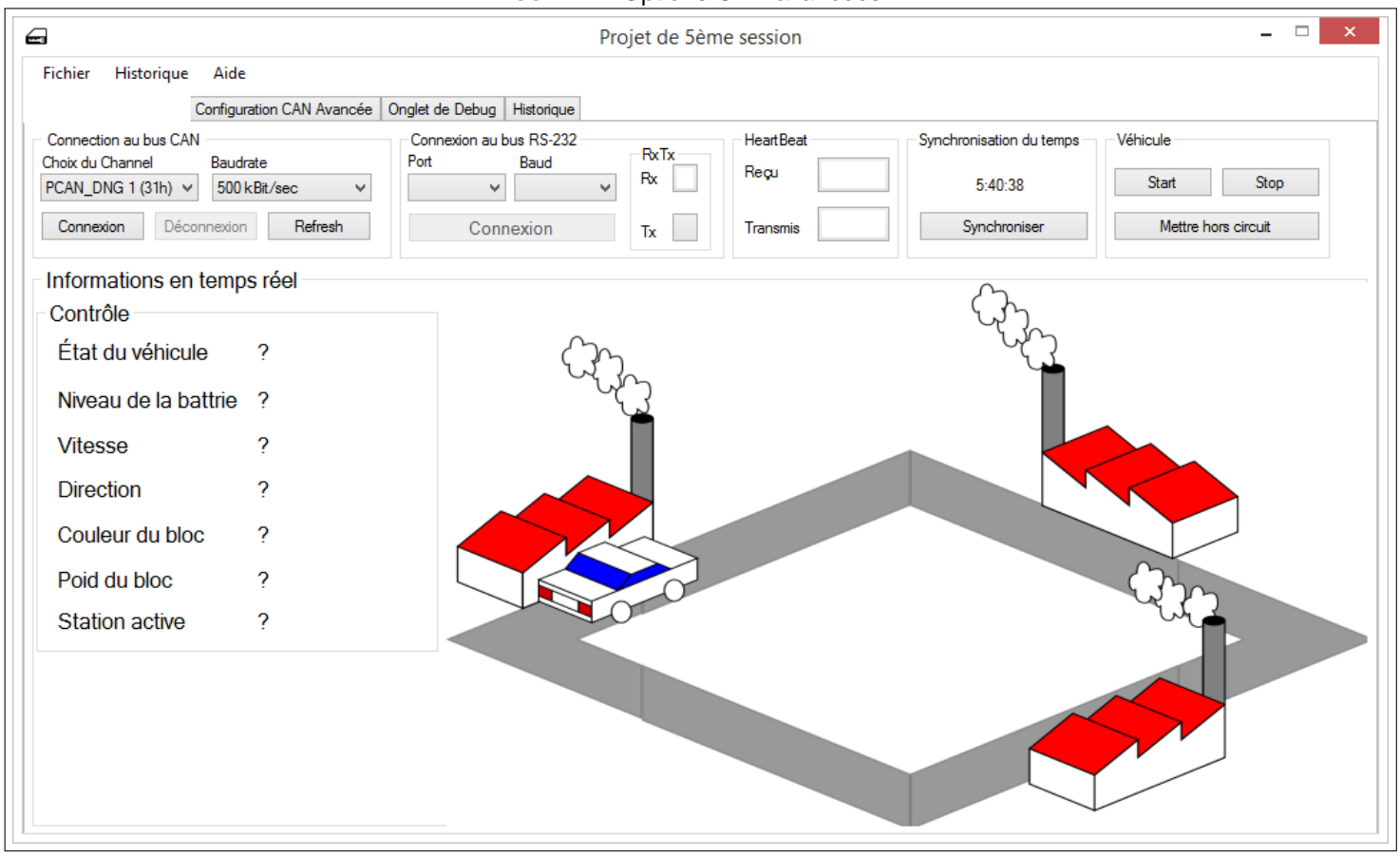


Notre programme, écrit en C# à l'aide de Visual Studio, peut se connecter au bus CAN via une carte SPI[2] et au bus RS232 via un câble DB9 ou USB[3]. La connexion RS232 sert à l'envoie et à la réception du HeartBeat afin d'informer le SOC8200 si l'ordinateur en venait à connaître une défaillance. De plus, des témoins lumineux s'allument en présence de données transmises et reçues.

Le programme peut lire l'heure interne du PC et, par un simple clic sur le bouton ≪Synchroniser≫, ajuster l'heure de référence de la station no.1.

2. Spécifier le fabricant
3. S'il y a présence d'un FTDI

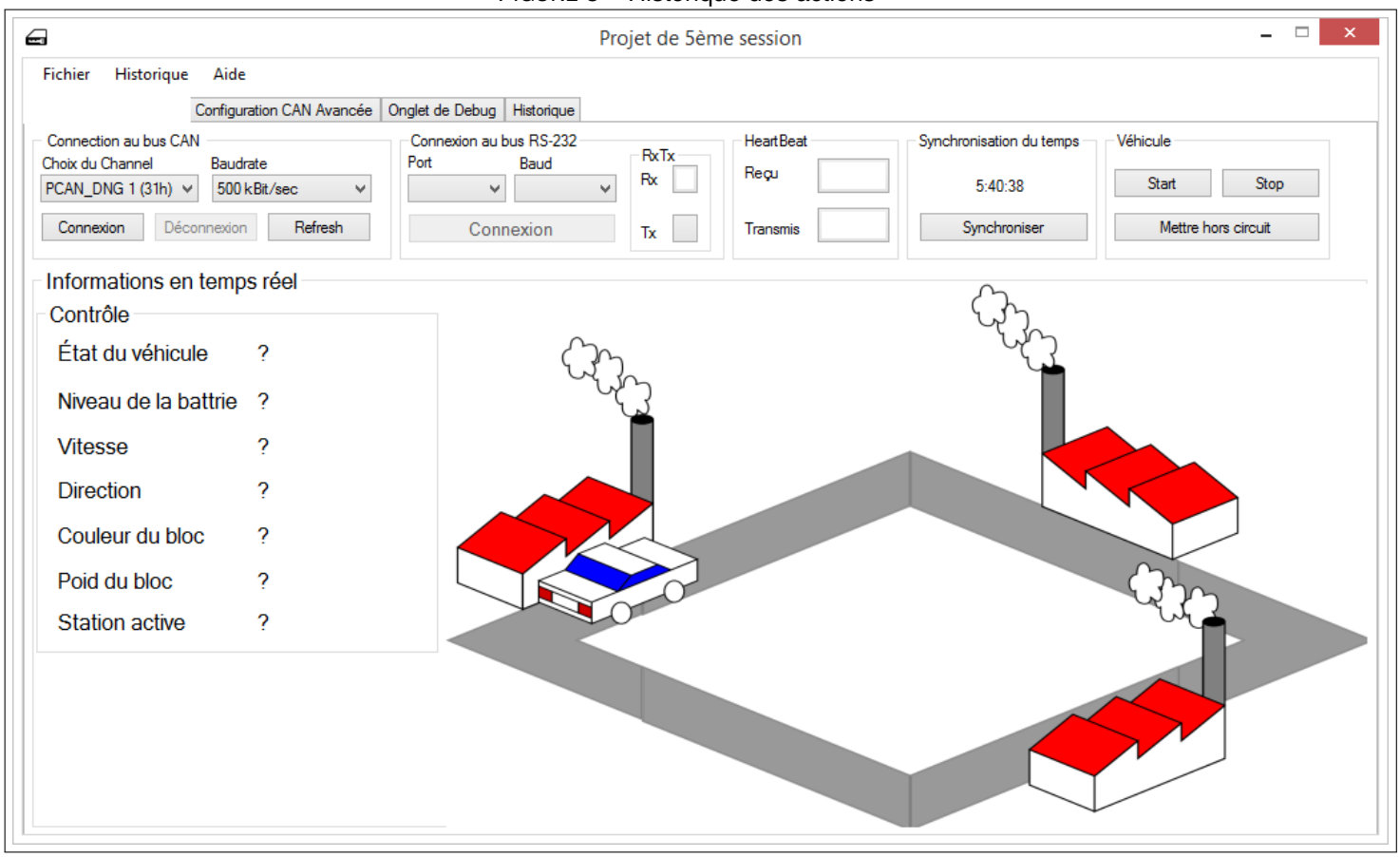
FIGURE 2 – Options CAN avancées



Il est possible d'utiliser des fonctionnalités CAN avancées tels que les masques et filtres de données. De plus, cette fenêtre permet de visualiser les données CAN reçues à l'état brutes et non traitées, ce qui peut s'avérer utile pour du débogage.

3.1 Gestion de l'historique

FIGURE 3 – Historique des actions



Toute action effectué via le programme ainsi que toute donnée ayant transité sur le bus CAN et RS232 est catalogué en bonne et due forme dans l'onglet historique qu'il est possible de consulter et sauvegarder à tout moment.

3.1.1 Exemple d'historique typique

```
wefsd
```

3.2 Structure du programme

3.2.1 Les Ghosts Labels

Un ghost label est un label de texte présent sur l'interface, mais définit comme invisible. Il est doc impossible pour l'usager de le voir et d'y accéder. Leurs principales utilités est de faire office de variable globale afin de passer des paramètres entre fonctions et de déclencher des événements système lorsqu'ils sont lut ou modifié.

3.3 Explication des trames

3.4 Ordre de gestion des tâches

4 Logiciel du SOC8200

4.1 Description du programme

Le programme du SOC2800 est écrit en script Shell [4] [5]

4.2 Schéma bloc

4.2.1 Du code

4.2.2 Du script shell

4.3 Gestion des processus et du temps de CPU

4.4 Format et récupration des logs

4.5 Liste des tests et logiciels

4. Car c'est plus simple que de se battre avec le gestionnaire de licence de Sourcery Codebench
5. L'un des membres de l'équipe fait dire que les licences sont une horreur inacceptable sur un système Linux

5 Logiciel de la station 1 et 2 et du bolide

Le programme de la station 1, 2 et du bolide est écrit en C++ à l'aide d'IAR WorkBench 8.20 et la compilation conditionnelle offre de le compiler pour chacune des trois stations mentionnées. De plus, la compilation conditionnelle permet au bolide d'utiliser soit une carte d'extension I2C, soit une carte d'extension SPI pour contrôler ses moteurs.

5.1 La station no.1

La sation no.1 est composé de μPSD et s'appelle Bloc no.2 dans le cahier de consignes. Cette station reçoit les directives du PC par le BUS CAN et les expédie sur le bus CAN (et vice-versa) aux endroits appropriés. C'est aussi à cette station qu'incombe la tâche de communiquer avec le bolide et la table FESTO via des Xbee

5.2 La station no.2

La sation no.2 est composé de la table FESTO et s'appelle Bloc no.3 dans le cahier de consignes

5.3 Le bolide

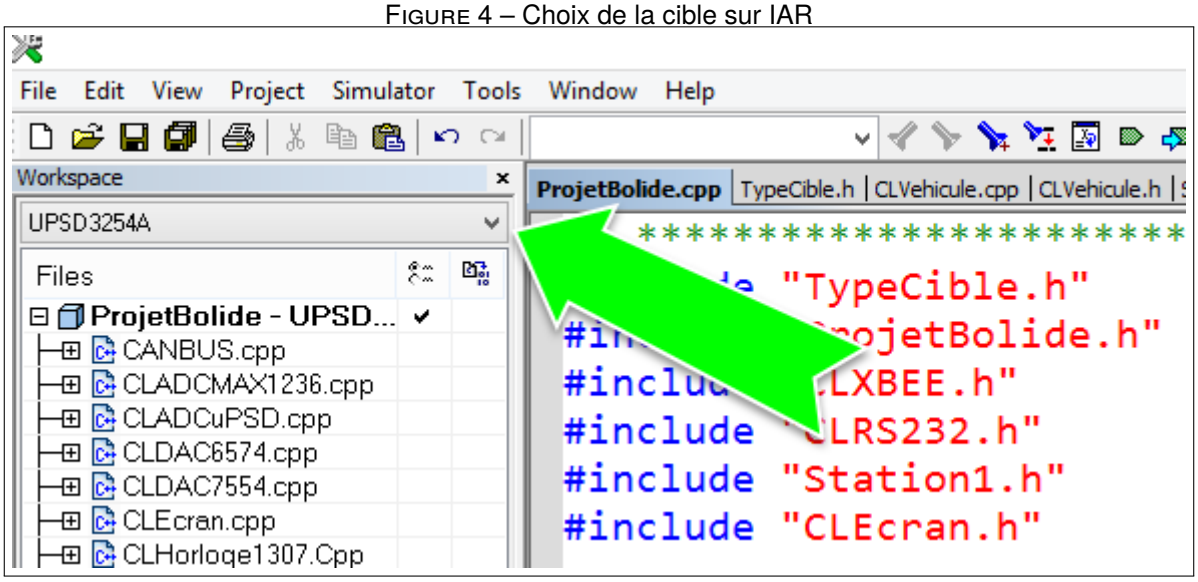
Composante	**Adresse I2C**	**Description**
MAX1236	0x68	Convertisseur analogique-numérique
DS1307	0xD0	Circuit d'horloge RTC
PCF8574	0x40	I/O Expender pour bus I2C
DAC6574	0x98	Convertisseur numérique-analogique
OPT101	0x50	Suiveur de ligne

TABLE 4 – Informations sur le bus I2C du bolide

5.4 Procédure de compilation sur IAR

Sur IAR, vous pouvez utiliser le menu déroulant, illustré à la figure suivante, afin de compiler le code pour la carte Dallas ou pour la carte μPSD.

FIGURE 4 – Choix de la cible sur IAR



De plus, des paramètres de compilation optionnelle vous permettent, via la décommentation, de compiler le code pour la carte Dallasou μPSD, pour la carte d'extension I2C ou SPI et pour un capteur de ligne à 3 ou à 5 photoréccepteurs.

Appercu des directives de compilation conditionnelles

```
//#define UPSD3254A
//#define DALLAS89C450
//#define SPI_DALLAS
//#define I2C_DALLAS
//#define PCF_5_CAPTEURS
//#define PCF_3_CAPTEURS
```

5.5 Procédure de vérification

6 Logiciel du module PIC18F258

6.1 Description du fonctionnement du programme

6.2 Procédure de compilation sur MPLAB

6.3 Procédure de vérification

7 Calculs

7.1 Calcul du pas de conversion de la pile

$$V_{MAX} = 10.8V \Rightarrow MAX1236 = 12bit \Rightarrow Pas = \frac{4K\Omega \cdot \left(\frac{10.8V}{10K\Omega}\right)}{2^{12}(pas)} = 1.33(mV/pas)$$

7.2 Calcul du baudrate

$$Baud = \frac{2^{SMOD}}{32} \cdot \frac{Crystal(Hz)}{12 \cdot (256 - TH1)}$$

Alors...

$$\overbrace{9600 = \frac{2^1}{32} \cdot \frac{24 \cdot 10^6(Hz)}{12 \cdot (256-243)}}^{uPSD3254} \Leftarrow \& \Rightarrow \overbrace{9600 = \frac{2^0}{32} \cdot \frac{11.0597 \cdot 10^6(Hz)}{12 \cdot (256-253)}}^{DS89C450}$$

8 Conclusion

8.1 Ce que le projet m'a apporté

8.1.1 Vincent Chouinard

8.1.2 Hicham Safoine

8.1.3 Gabriel Fortin-Bélanger

8.1.4 Louis-Norman Ang-Houle

8.2 Difficultés et corrections

8.2.1 Vincent Chouinard

8.2.2 Hicham Safoine

8.2.3 Gabriel Fortin-Bélanger

8.2.4 Louis-Norman Ang-Houle

8.3 Ce que j'ai aimé ou pas

8.3.1 Vincent Chouinard

8.3.2 Hicham Safoine

8.3.3 Gabriel Fortin-Bélanger

8.3.4 Louis-Norman Ang-Houle

9 ANNEXE 1 : Code source du Bolide et de la station no.1

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

/// <summary>
/// Inclusion of PEAK PCAN-Basic namespace
/// </summary>
using Peak.Can.Basic;
using TPCANHandle = System.Byte;

namespace ICDIBasic
{
    public partial class Form1 : Form
    {
        #region Structures
        /// <summary>
        /// Message Status structure used to show CAN Messages
        /// in a ListView
        /// </summary>
        private class MessageStatus
        {
            private TPCANMsg m_Msg;
            private TPCANTimestamp m_TimeStamp;
            private TPCANTimestamp m_oldTimeStamp;
            private int m_iIndex;
            private int m_Count;
            private bool m_bShowPeriod;
            private bool m_bWasChanged;

            public MessageStatus(TPCANMsg canMsg, TPCANTimestamp canTimestamp, int listIndex)
            {
                m_Msg = canMsg;
                m_TimeStamp = canTimestamp;
                m_oldTimeStamp = canTimestamp;
                m_iIndex = listIndex;
                m_Count = 1;
                m_bShowPeriod = true;
                m_bWasChanged = false;
            }

            public void Update(TPCANMsg canMsg, TPCANTimestamp canTimestamp)
            {
                m_Msg = canMsg;
                m_oldTimeStamp = m_TimeStamp;
                m_TimeStamp = canTimestamp;
                m_bWasChanged = true;
                m_Count += 1;
            }

            public TPCANMsg CANMsg
            {
                get { return m_Msg; }
            }

            public TPCANTimestamp Timestamp
            {
                get { return m_TimeStamp; }
            }

            public int Position
            {
                get { return m_iIndex; }
            }

            public string TypeString
            {
                get { return GetMsgTypeString(); }
            }

            public string IdString
```

```
        {
            get { return GetIdString(); }
        }

        public string DataString
        {
            get { return GetDataString(); }
        }

        public int Count
        {
            get { return m_Count; }
        }

        public bool ShowingPeriod
        {
            get { return m_bShowPeriod; }
            set
            {
                if (m_bShowPeriod ^ value)
                {
                    m_bShowPeriod = value;
                    m_bWasChanged = true;
                }
            }
        }

        public bool MarkedAsUpdated
        {
            get { return m_bWasChanged; }
            set { m_bWasChanged = value; }
        }

        public string TimeString
        {
            get { return GetTimeString(); }
        }

        private string GetTimeString()
        {
            double fTime;

            fTime = m_TimeStamp.millis + (m_TimeStamp.micros / 1000.0);
            if (m_bShowPeriod)
                fTime -= (m_oldTimeStamp.millis + (m_oldTimeStamp.micros / 1000.0));
            return fTime.ToString("F1");
        }

        private string GetDataString()
        {
            string strTemp;

            strTemp = "";

            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_RTR) == TPCANMessageType.PCAN_MESSAGE_RTR)
                return "Remote Request";
            else
                for (int i = 0; i < m_Msg.LEN; i++)
                    strTemp += string.Format("{0:X2} ", m_Msg.DATA[i]);

            return strTemp;
        }

        private string GetIdString()
        {
            // We format the ID of the message and show it
            //
            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_EXTENDED) == TPCANMessageType.
PCAN_MESSAGE_EXTENDED)
                return string.Format("{0:X8}h", m_Msg.ID);
            else
                return string.Format("{0:X3}h", m_Msg.ID);
        }

        private string GetMsgTypeString()
        {
```

```
            string strTemp;

            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_EXTENDED) == TPCANMessageType.
PCAN_MESSAGE_EXTENDED)
                strTemp = "EXTENDED";
            else
                strTemp = "STANDARD";

            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_RTR) == TPCANMessageType.PCAN_MESSAGE_RTR)
                strTemp += "/RTR";

            return strTemp;
        }

    }
    #endregion

    #region Delegates
    /// <summary>
    /// Read-Delegate Handler
    /// </summary>
    private delegate void ReadDelegateHandler();
    #endregion

    #region Members
    /// <summary>
    /// Saves the handle of a PCAN hardware
    /// </summary>
    private TPCANHandle m_PcanHandle;
    /// <summary>
    /// Saves the baudrate register for a conenction
    /// </summary>
    private TPCANBaudrate m_Baudrate;
    /// <summary>
    /// Saves the type of a non-plug-and-play hardware
    /// </summary>
    private TPCANType m_HwType;
    /// <summary>
    /// Stores the status of received messages for its display
    /// </summary>
    private System.Collections.ArrayList m_LastMsgsList;
    /// <summary>
    /// Read Delegate for calling the function "ReadMessages"
    /// </summary>
    private ReadDelegateHandler m_ReadDelegate;
    /// <summary>
    /// Receive-Event
    /// </summary>
    private System.Threading.AutoResetEvent m_ReceiveEvent;
    /// <summary>
    /// Thread for message reading (using events)
    /// </summary>
    private System.Threading.Thread m_ReadThread;
    /// <summary>
    /// Handles of the current available PCAN-Hardware
    /// </summary>
    private TPCANHandle[] m_HandlesArray;
    #endregion

    #region Methods
    #region Help functions
    /// <summary>
    /// Initialization of PCAN-Basic components
    /// </summary>
    private void InitializeBasicComponents()
    {
        // Creates the list for received messages
        //
        m_LastMsgsList = new System.Collections.ArrayList();
        // Creates the delegate used for message reading
        //
        m_ReadDelegate = new ReadDelegateHandler(ReadMessages);
        // Creates the event used for signalize incomming messages
        //
        m_ReceiveEvent = new System.Threading.AutoResetEvent(false);
        // Creates an array with all possible PCAN-Channels
```

```
            //
            m_HandlesArray = new TPCANHandle[]
            {
                PCANBasic.PCAN_ISABUS1,
                PCANBasic.PCAN_ISABUS2,
                PCANBasic.PCAN_ISABUS3,
                PCANBasic.PCAN_ISABUS4,
                PCANBasic.PCAN_ISABUS5,
                PCANBasic.PCAN_ISABUS6,
                PCANBasic.PCAN_ISABUS7,
                PCANBasic.PCAN_ISABUS8,
                PCANBasic.PCAN_DNGBUS1,
                PCANBasic.PCAN_PCIBUS1,
                PCANBasic.PCAN_PCIBUS2,
                PCANBasic.PCAN_PCIBUS3,
                PCANBasic.PCAN_PCIBUS4,
                PCANBasic.PCAN_PCIBUS5,
                PCANBasic.PCAN_PCIBUS6,
                PCANBasic.PCAN_PCIBUS7,
                PCANBasic.PCAN_PCIBUS8,
                PCANBasic.PCAN_USBBUS1,
                PCANBasic.PCAN_USBBUS2,
                PCANBasic.PCAN_USBBUS3,
                PCANBasic.PCAN_USBBUS4,
                PCANBasic.PCAN_USBBUS5,
                PCANBasic.PCAN_USBBUS6,
                PCANBasic.PCAN_USBBUS7,
                PCANBasic.PCAN_USBBUS8,
                PCANBasic.PCAN_PCCBUS1,
                PCANBasic.PCAN_PCCBUS2
            };

            // Fills and configures the Data of several comboBox components
            //
            FillComboBoxData();

            // Prepares the PCAN-Basic's debug-Log file
            //
            ConfigureLogFile();
        }

        /// <summary>
        /// COnfigures the Debug-Log file of PCAN-Basic
        /// </summary>
        private void ConfigureLogFile()
        {
            UInt32 iBuffer;

            // Sets the mask to catch all events
            //
            iBuffer = PCANBasic.LOG_FUNCTION_ENTRY | PCANBasic.LOG_FUNCTION_LEAVE | PCANBasic.
LOG_FUNCTION_PARAMETERS |
                PCANBasic.LOG_FUNCTION_READ | PCANBasic.LOG_FUNCTION_WRITE;

            // Configures the log file.
            // NOTE: The Log capability is to be used with the NONEBUS Handle. Other handle than this will
            // cause the function fail.
            //
            PCANBasic.SetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_LOG_CONFIGURE, ref iBuffer, sizeof(
UInt32));
        }

        /// <summary>
        /// Help Function used to get an error as text
        /// </summary>
        /// <param name="error">Error code to be translated</param>
        /// <returns>A text with the translated error</returns>
        private string GetFormatedError(TPCANStatus error)
        {
            StringBuilder strTemp;

            // Creates a buffer big enough for a error-text
            //
            strTemp = new StringBuilder(256);
            // Gets the text using the GetErrorText API function
            // If the function success, the translated error is returned. If it fails,
```

```
            // a text describing the current error is returned.
            //
            if (PCANBasic.GetErrorText(error, 0, strTemp) != TPCANStatus.PCAN_ERROR_OK)
                return string.Format("An error occurred. Error-code's text ({0:X}) couldn't be retrieved", error
);
            else
                return strTemp.ToString();
        }

        /// <summary>
        /// Includes a new line of text into the information Listview
        /// </summary>
        /// <param name="strMsg">Text to be included</param>
        private void IncludeTextMessage(string strMsg)
        {
            lbxInfo.Items.Add(strMsg);
            lbxInfo.SelectedIndex = lbxInfo.Items.Count - 1;
        }

        /// <summary>
        /// Gets the current status of the PCAN-Basic message filter
        /// </summary>
        /// <param name="status">Buffer to retrieve the filter status</param>
        /// <returns>If calling the function was successfull or not</returns>
        private bool GetFilterStatus(out uint status)
        {
            TPCANStatus stsResult;

            // Tries to get the sttaus of the filter for the current connected hardware
            //
            stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_MESSAGE_FILTER, out status, sizeof(
UInt32));

            // If it fails, a error message is shown
            //
            if (stsResult != TPCANStatus.PCAN_ERROR_OK)
            {
                MessageBox.Show(GetFormatedError(stsResult));
                return false;
            }
            return true;
        }

        /// <summary>
        /// Configures the data of all ComboBox components of the main-form
        /// </summary>
        private void FillComboBoxData()
        {
            // Channels will be check
            //
            btnHwRefresh_Click(this, new EventArgs());

            // Baudrates
            //
            cbbBaudrates.SelectedIndex = 2; // 500 K

            // Hardware Type for no plugAndplay hardware
            //
            cbbHwType.SelectedIndex = 0;

            // Interrupt for no plugAndplay hardware
            //
            cbbInterrupt.SelectedIndex = 0;

            // IO Port for no plugAndplay hardware
            //
            cbbIO.SelectedIndex = 0;

            // Parameters for GetValue and SetValue function calls
            //
            cbbParameter.SelectedIndex = 0;
        }

        /// <summary>
        /// Activates/deaactivates the different controls of the main-form according
        /// with the current connection status
```

```
        /// </summary>
        /// <param name="bConnected">Current status. True if connected, false otherwise</param>
        private void SetConnectionStatus(bool bConnected)
        {
            // Buttons
            //
            btnInit.Enabled = !bConnected;
            btnRead.Enabled = bConnected && rdbManual.Checked;
            btnWrite.Enabled = bConnected;
            btnRelease.Enabled = bConnected;
            btnFilterApply.Enabled = bConnected;
            btnFilterQuery.Enabled = bConnected;
            btnParameterSet.Enabled = bConnected;
            btnParameterGet.Enabled = bConnected;
            btnGetVersions.Enabled = bConnected;
            btnHwRefresh.Enabled = !bConnected;
            btnStatus.Enabled = bConnected;
            btnReset.Enabled = bConnected;

            // ComboBoxs
            //
            cbbBaudrates.Enabled = !bConnected;
            cbbChannel.Enabled = !bConnected;
            cbbHwType.Enabled = !bConnected;
            cbbIO.Enabled = !bConnected;
            cbbInterrupt.Enabled = !bConnected;

            // Hardware configuration and read mode
            //
            if (!bConnected)
                cbbChannel_SelectedIndexChanged(this, new EventArgs());
            else
                rdbTimer_CheckedChanged(this, new EventArgs());

            // Display messages in grid
            //
            tmrDisplay.Enabled = bConnected;
        }

        /// <summary>
        /// Gets the formated text for a CPAN-Basic channel handle
        /// </summary>
        /// <param name="handle">PCAN-Basic Handle to format</param>
        /// <returns>The formatted text for a channel</returns>
        private string FormatChannelName(TPCANHandle handle)
        {
            TPCANDevice devDevice;
            byte byChannel;

            // Gets the owner device and channel for a
            // PCAN-Basic handle
            //
            devDevice = (TPCANDevice)(handle >> 4);
            byChannel = (byte)(handle & 0xF);

            // Constructs the PCAN-Basic Channel name and return it
            //
            return string.Format("{0} {1} ({2:X2}h)", devDevice, byChannel, handle);
        }
        #endregion

        #region Message-proccessing functions
        /// <summary>
        /// Display CAN messages in the Message-ListView
        /// </summary>
        private void DisplayMessages()
        {
            ListViewItem lviCurrentItem;

            lock (m_LastMsgsList.SyncRoot)
            {
                foreach (MessageStatus msgStatus in m_LastMsgsList)
                {
                    // Get the data to actualize
                    //
                    if (msgStatus.MarkedAsUpdated)
```

```
                {
                    msgStatus.MarkedAsUpdated = false;
                    lviCurrentItem = lstMessages.Items[msgStatus.Position];

                    lviCurrentItem.SubItems[2].Text = msgStatus.CANMsg.LEN.ToString();
                    lviCurrentItem.SubItems[3].Text = msgStatus.DataString;
                    lviCurrentItem.SubItems[4].Text = msgStatus.Count.ToString();
                    lviCurrentItem.SubItems[5].Text = msgStatus.TimeString;
                }
            }
        }
    }

    /// <summary>
    /// Inserts a new entry for a new message in the Message-ListView
    /// </summary>
    /// <param name="newMsg">The messasge to be inserted</param>
    /// <param name="timeStamp">The Timesamp of the new message</param>
    private void InsertMsgEntry(TPCANMsg newMsg, TPCANTimestamp timeStamp)
    {
        MessageStatus msgStsCurrentMsg;
        ListViewItem lviCurrentItem;

        lock (m_LastMsgsList.SyncRoot)
        {
            // We add this status in the last message list
            //
            msgStsCurrentMsg = new MessageStatus(newMsg, timeStamp, lstMessages.Items.Count);
            m_LastMsgsList.Add(msgStsCurrentMsg);

            // Add the new ListView Item with the Type of the message
            //
            lviCurrentItem = lstMessages.Items.Add(msgStsCurrentMsg.TypeString);
            // We set the ID of the message
            //
            lviCurrentItem.SubItems.Add(msgStsCurrentMsg.IdString);
            // We set the length of the Message
            //
            lviCurrentItem.SubItems.Add(newMsg.LEN.ToString());
            // We set the data of the message.
            //
            lviCurrentItem.SubItems.Add(msgStsCurrentMsg.DataString);
            // we set the message count message (this is the First, so count is 1)
            //
            lviCurrentItem.SubItems.Add(msgStsCurrentMsg.Count.ToString());
            // Add time stamp information if needed
            //
            lviCurrentItem.SubItems.Add(msgStsCurrentMsg.TimeString);
        }
    }

    /// <summary>
    /// Processes a received message, in order to show it in the Message-ListView
    /// </summary>
    /// <param name="theMsg">The received PCAN-Basic message</param>
    /// <returns>True if the message must be created, false if it must be modified</returns>
    private void ProcessMessage(TPCANMsg theMsg, TPCANTimestamp itsTimeStamp)
    {
        // We search if a message (Same ID and Type) is
        // already received or if this is a new message
        //
        lock (m_LastMsgsList.SyncRoot)
        {
            foreach (MessageStatus msg in m_LastMsgsList)
            {
                if ((msg.CANMsg.ID == theMsg.ID) && (msg.CANMsg.MSGTYPE == theMsg.MSGTYPE))
                {
                    // Modify the message and exit
                    //
                    msg.Update(theMsg, itsTimeStamp);
                    return;
                }
            }
            // Message not found. It will created
            //
            InsertMsgEntry(theMsg, itsTimeStamp);
```

```
            }
        }

        /// <summary>
        /// Thread-Function used for reading PCAN-Basic messages
        /// </summary>
        private void CANReadThreadFunc()
        {
            UInt32 iBuffer;
            TPCANStatus stsResult;

            iBuffer = Convert.ToUInt32(m_ReceiveEvent.SafeWaitHandle.DangerousGetHandle().ToInt32());
            // Sets the handle of the Receive-Event.
            //
            stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_RECEIVE_EVENT, ref iBuffer, sizeof(
UInt32));

            if(stsResult != TPCANStatus.PCAN_ERROR_OK)
          {
             MessageBox.Show(GetFormatedError(stsResult),"Error!",MessageBoxButtons.OK,MessageBoxIcon.Error);
             return;
          }

            // While this mode is selected
            while (rdbEvent.Checked)
            {
                // Waiting for Receive-Event
                //
                if (m_ReceiveEvent.WaitOne(50))
                    // Process Receive-Event using .NET Invoke function
                    // in order to interact with Winforms UI (calling the
                    // function ReadMessages)
                    //
                    this.Invoke(m_ReadDelegate);
            }
        }

        /// <summary>
        /// Function for reading PCAN-Basic messages
        /// </summary>
        private void ReadMessages()
        {
            TPCANMsg CANMsg;
            TPCANTimestamp CANTimeStamp;
            TPCANStatus stsResult;

            // We read at least one time the queue looking for messages.
            // If a message is found, we look again trying to find more.
            // If the queue is empty or an error occurr, we get out from
            // the dowhile statement.
            //
            do
            {
                // We execute the "Read" function of the PCANBasic
                //
                stsResult = PCANBasic.Read(m_PcanHandle, out CANMsg, out CANTimeStamp);

                // A message was received
                // We process the message(s)
                //
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    ProcessMessage(CANMsg, CANTimeStamp);

            } while (btnRelease.Enabled && (!Convert.ToBoolean(stsResult & TPCANStatus.PCAN_ERROR_QRCVEMPTY)));
        }
        #endregion

        #region Event Handlers
        #region Form event-handlers
        /// <summary>
        /// Consturctor
        /// </summary>
        public Form1()
        {
            // Initializes Form's component
            //
```

```
        InitializeComponent();
        // Initializes specific components
        //
        InitializeBasicComponents();
    }

    /// <summary>
    /// Form-Closing Function / Finish function
    /// </summary>
    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
        // Releases the used PCAN-Basic channel
        //
        PCANBasic.Uninitialize(m_PcanHandle);
    }
    #endregion

    #region ComboBox event-handlers
    private void cbbChannel_SelectedIndexChanged(object sender, EventArgs e)
    {
        bool bNonPnP;
        string strTemp;

        // Get the handle fromt he text being shown
        //
        strTemp = cbbChannel.Text;
        strTemp = strTemp.Substring(strTemp.IndexOf('(')+1, 2);

        // Determines if the handle belong to a No Plug&Play hardware
        //
        m_PcanHandle = Convert.ToByte(strTemp,16);
        bNonPnP = m_PcanHandle <= PCANBasic.PCAN_DNGBUS1;
        // Activates/deactivates configuration controls according with the
        // kind of hardware
        //
        cbbHwType.Enabled = bNonPnP;
        cbbIO.Enabled = bNonPnP;
        cbbInterrupt.Enabled = bNonPnP;
    }

    private void cbbBaudrates_SelectedIndexChanged(object sender, EventArgs e)
    {
        // Saves the current selected baudrate register code
        //
        switch (cbbBaudrates.SelectedIndex)
        {
            case 0:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_1M;
                break;
            case 1:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_800K;
                break;
            case 2:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_500K;
                break;
            case 3:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_250K;
                break;
            case 4:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_125K;
                break;
            case 5:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_100K;
                break;
            case 6:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_95K;
                break;
            case 7:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_83K;
                break;
            case 8:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_50K;
                break;
            case 9:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_47K;
                break;
```

```
            case 10:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_33K;
                break;
            case 11:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_20K;
                break;
            case 12:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_10K;
                break;
            case 13:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_5K;
                break;
        }
    }

    private void cbbHwType_SelectedIndexChanged(object sender, EventArgs e)
    {
        // Saves the current type for a no-Plug&Play hardware
        //
        switch (cbbHwType.SelectedIndex)
        {
            case 0:
                m_HwType = TPCANType.PCAN_TYPE_ISA;
                break;
            case 1:
                m_HwType = TPCANType.PCAN_TYPE_ISA_SJA;
                break;
            case 2:
                m_HwType = TPCANType.PCAN_TYPE_ISA_PHYTEC;
                break;
            case 3:
                m_HwType = TPCANType.PCAN_TYPE_DNG;
                break;
            case 4:
                m_HwType = TPCANType.PCAN_TYPE_DNG_EPP;
                break;
            case 5:
                m_HwType = TPCANType.PCAN_TYPE_DNG_SJA;
                break;
            case 6:
                m_HwType = TPCANType.PCAN_TYPE_DNG_SJA_EPP;
                break;
        }
    }

    private void cbbParameter_SelectedIndexChanged(object sender, EventArgs e)
    {
        // Activates/deactivates controls according with the selected
        // PCAN-Basic parameter
        //
        rdbParamActive.Enabled = cbbParameter.SelectedIndex != 0;
        rdbParamInactive.Enabled = rdbParamActive.Enabled;
        nudDeviceId.Enabled = !rdbParamActive.Enabled;
    }
    #endregion

    #region Button event-handlers
    private void btnHwRefresh_Click(object sender, EventArgs e)
    {
        UInt32 iBuffer;
        TPCANStatus stsResult;

        // Clears the Channel combioBox and fill it again with
        // the PCAN-Basic handles for no-Plug&Play hardware and
        // the detected Plug&Play hardware
        //
        cbbChannel.Items.Clear();
        try
        {
            for (int i = 0; i < m_HandlesArray.Length; i++)
            {
                // Includes all no-Plug&Play Handles
                if (m_HandlesArray[i] <= PCANBasic.PCAN_DNGBUS1)
                    cbbChannel.Items.Add(FormatChannelName(m_HandlesArray[i]));
                else
                {
```

```
                    // Checks for a Plug&Play Handle and, according with the return value, includes it
                    // into the list of available hardware channels.
                    //
                    stsResult = PCANBasic.GetValue(m_HandlesArray[i], TPCANParameter.PCAN_CHANNEL_CONDITION,
 out iBuffer, sizeof(UInt32));
                    if ((stsResult == TPCANStatus.PCAN_ERROR_OK) && (iBuffer == PCANBasic.
PCAN_CHANNEL_AVAILABLE))
                        cbbChannel.Items.Add(FormatChannelName(m_HandlesArray[i]));
                }
            }
            cbbChannel.SelectedIndex = cbbChannel.Items.Count - 1;
        }
        catch (DllNotFoundException)
        {
            MessageBox.Show("Le fichier PCANBasic.dll est introuvable!", "Missing DLL Err0r!",
MessageBoxButtons.OK, MessageBoxIcon.Error);
            Environment.Exit(-1);
        }
    }

    private void btnInit_Click(object sender, EventArgs e)
    {
        TPCANStatus stsResult;

        // Connects a selected PCAN-Basic channel
        //
        stsResult = PCANBasic.Initialize(
            m_PcanHandle,
            m_Baudrate,
            m_HwType,
            Convert.ToUInt32(cbbIO.Text,16),
            Convert.ToUInt16(cbbInterrupt.Text));

        if (stsResult != TPCANStatus.PCAN_ERROR_OK)
            MessageBox.Show(GetFormatedError(stsResult));

        // Sets the connection status of the main-form
        //
        SetConnectionStatus(stsResult == TPCANStatus.PCAN_ERROR_OK);
    }

    private void btnRelease_Click(object sender, EventArgs e)
    {
        // Releases a current connected PCAN-Basic channel
        //
        PCANBasic.Uninitialize(m_PcanHandle);
        tmrRead.Enabled = false;
        if (m_ReadThread != null)
        {
            m_ReadThread.Abort();
            m_ReadThread.Join();
            m_ReadThread = null;
        }

        // Sets the connection status of the main-form
        //
        SetConnectionStatus(false);
    }

    private void btnFilterApply_Click(object sender, EventArgs e)
    {
        UInt32 iBuffer;
        TPCANStatus stsResult;

        // Gets the current status of the message filter
        //
        if (!GetFilterStatus(out iBuffer))
            return;

        // Configures the message filter for a custom range of messages
        //
        if (rdbFilterCustom.Checked)
        {
            // The filter must be first closed in order to customize it
            //
            if (iBuffer != PCANBasic.PCAN_FILTER_OPEN)
```

```
            {
                // Sets the custom filter
                //
                stsResult = PCANBasic.FilterMessages(
                m_PcanHandle,
                Convert.ToUInt32(nudIdFrom.Value),
                Convert.ToUInt32(nudIdTo.Value),
                chbFilterExt.Checked ? TPCANMode.PCAN_MODE_EXTENDED : TPCANMode.PCAN_MODE_STANDARD);
                // If success, an information message is written, if it is not, an error message is shown
                //
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The filter was customized. IDs from {0:X} to {1:X}
will be received", nudIdFrom.Text, nudIdTo.Text));
                else
                    MessageBox.Show(GetFormatedError(stsResult));
            }
            else
                MessageBox.Show("Le filtre doit être fermé si vous voulez le modifier");

            return;
        }

        // The filter will be full opened or complete closed
        //
        if (rdbFilterClose.Checked)
            iBuffer = PCANBasic.PCAN_FILTER_CLOSE;
        else
            iBuffer = PCANBasic.PCAN_FILTER_OPEN;

        // The filter is configured
        //
        stsResult = PCANBasic.SetValue(
            m_PcanHandle,
            TPCANParameter.PCAN_MESSAGE_FILTER,
            ref iBuffer,
            sizeof(UInt32));

        // If success, an information message is written, if it is not, an error message is shown
        //
        if (stsResult == TPCANStatus.PCAN_ERROR_OK)
           IncludeTextMessage(string.Format("The filter a été {0}", rdbFilterClose.Checked ? "closed." : "
opened."));
        else
            MessageBox.Show(GetFormatedError(stsResult));
    }

    private void btnFilterQuery_Click(object sender, EventArgs e)
    {
        UInt32 iBuffer;

        // Queries the current status of the message filter
        //
        if (GetFilterStatus(out iBuffer))
        {
            switch(iBuffer)
            {
                // The filter is closed
                //
                case  PCANBasic.PCAN_FILTER_CLOSE:
                    IncludeTextMessage("Le filtre est: fermé.");
                    break;
                // The filter is fully opened
                //
                case PCANBasic.PCAN_FILTER_OPEN:
                    IncludeTextMessage("Le filtre est: ouvert.");
                    break;
                // The filter is customized
                //
                case PCANBasic.PCAN_FILTER_CUSTOM:
                    IncludeTextMessage("Le filtre est: personnalisé.");
                    break;
                // The status of the filter is undefined. (Should never happen)
                //
                default:
                    IncludeTextMessage("Le filtre est: invalide.");
                    break;
```

```
            }
        }
    }

    private void btnParameterSet_Click(object sender, EventArgs e)
    {
        TPCANStatus stsResult;
        UInt32 iBuffer;

        // Sets a PCAN-Basic parameter value
        //
        switch (cbbParameter.SelectedIndex)
        {
            // The Device-Number of an USB channel will be set
            //
            case 0:
                iBuffer = Convert.ToUInt32(nudDeviceId.Value);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_DEVICE_NUMBER, ref iBuffer,
 sizeof(UInt32));
                if(stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage("The desired Device-Number was successfully configured");
                break;
            // The 5 Volt Power feature of a PC-card or USB will be set
            //
            case 1:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_5VOLTS_POWER, ref iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The USB/PC-Card 5 power was successfully {0}", (
iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The feature for automatic reset on BUS-OFF will be set
            //
            case 2:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_BUSOFF_AUTORESET, ref
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The automatic-reset on BUS-OFF was successfully {0}",
(iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The CAN option "Listen Only" will be set
            //
            case 3:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_LISTEN_ONLY, ref iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The CAN-option Listen-Only was successfully {0}", (
iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The feature for logging debug-information will be set
            //
            case 4:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_LOG_STATUS, ref
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The feature for logging debug information was
successfully {0}", (iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The current parameter is invalid
            //
            default:
                stsResult = TPCANStatus.PCAN_ERROR_UNKNOWN;
                MessageBox.Show("Wrong parameter code.");
                return;
        }

        // If the function fail, an error message is shown
        //
```

```
        if(stsResult != TPCANStatus.PCAN_ERROR_OK)
            MessageBox.Show(GetFormatedError(stsResult));
    }

    private void btnParameterGet_Click(object sender, EventArgs e)
    {
        TPCANStatus stsResult;
        UInt32 iBuffer;

        // Gets a PCAN-Basic parameter value
        //
        switch (cbbParameter.SelectedIndex)
        {
            // The Device-Number of an USB channel will be retrieved
            //
            case 0:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_DEVICE_NUMBER, out iBuffer,
 sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The configured Device-Number is {0:X}h",iBuffer));
                break;
            // The activation status of the 5 Volt Power feature of a PC-card or USB will be retrieved
            //
            case 1:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_5VOLTS_POWER, out iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The 5-Volt Power of the USB/PC-Card is {0}", (iBuffer
== PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The activation status of the feature for automatic reset on BUS-OFF will be retrieved
            //
            case 2:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_BUSOFF_AUTORESET, out
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The automatic-reset on BUS-OFF is {0}", (iBuffer ==
PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The activation status of the CAN option "Listen Only" will be retrieved
            //
            case 3:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_LISTEN_ONLY, out iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The CAN-option Listen-Only is {0}", (iBuffer ==
PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The activation status for the feature for logging debug-information will be retrieved
            case 4:
                stsResult = PCANBasic.GetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_LOG_STATUS, out
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The feature for logging debug information is {0}", (
iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The current parameter is invalid
            //
            default:
                stsResult = TPCANStatus.PCAN_ERROR_UNKNOWN;
                MessageBox.Show("Wrong parameter code.");
                return;
        }

        // If the function fail, an error message is shown
        //
        if (stsResult != TPCANStatus.PCAN_ERROR_OK)
            MessageBox.Show(GetFormatedError(stsResult));
    }

    private void btnRead_Click(object sender, EventArgs e)
    {
        TPCANMsg CANMsg;
        TPCANTimestamp CANTimeStamp;
        TPCANStatus stsResult;
```

```
            // We execute the "Read" function of the PCANBasic
            //
            stsResult = PCANBasic.Read(m_PcanHandle, out CANMsg, out CANTimeStamp);
            if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                // We process the received message
                //
                ProcessMessage(CANMsg, CANTimeStamp);
            else
                // If an error occurred, an information message is included
                //
                IncludeTextMessage(GetFormatedError(stsResult));
        }

        private void btnGetVersions_Click(object sender, EventArgs e)
        {
            TPCANStatus stsResult;
            StringBuilder strTemp;
            string[] strArrayVersion;

            strTemp = new StringBuilder(256);

            // We get the vesion of the PCAN-Basic API
            //
            stsResult = PCANBasic.GetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_API_VERSION, strTemp,
256);
            if (stsResult == TPCANStatus.PCAN_ERROR_OK)
            {
                IncludeTextMessage("API Version: " + strTemp.ToString());
                // We get the driver version of the channel being used
                //
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_CHANNEL_VERSION, strTemp, 256);
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                {
                    // Because this information contains line control characters (several lines)
                    // we split this also in several entries in the Information List-Box
                    //
                    strArrayVersion = strTemp.ToString().Split(new char[] { '\n' });
                    IncludeTextMessage("Channel/Driver Version: ");
                    for(int i =0; i < strArrayVersion.Length; i++)
                        IncludeTextMessage("     * " + strArrayVersion[i]);
                }
            }

            // If an error ccurred, a message is shown
            //
            if (stsResult != TPCANStatus.PCAN_ERROR_OK)
                MessageBox.Show(GetFormatedError(stsResult));
        }

        private void btnMsgClear_Click(object sender, EventArgs e)
        {
            // The information contained in the messages List-View
            // is cleared
            //
            lock (m_LastMsgsList.SyncRoot)
            {
                m_LastMsgsList.Clear();
                lstMessages.Items.Clear();
            }
        }

        private void btnInfoClear_Click(object sender, EventArgs e)
        {
            // The information contained in the Information List-Box
            // is cleared
            //
            lbxInfo.Items.Clear();
        }

        private void btnWrite_Click(object sender, EventArgs e)
        {
            TPCANMsg CANMsg;
            TextBox txtbCurrentTextBox;
            TPCANStatus stsResult;

            // We create a TPCANMsg message structure
```

```
            //
            CANMsg = new TPCANMsg();
            CANMsg.DATA = new byte[8];

            // We configurate the Message.  The ID (max 0x1FF),
            // Length of the Data, Message Type (Standard in
            // this example) and die data
            //
            CANMsg.ID = Convert.ToUInt32(txtID.Text, 16);
            CANMsg.LEN = Convert.ToByte(nudLength.Value);
            CANMsg.MSGTYPE = (chbExtended.Checked) ? TPCANMessageType.PCAN_MESSAGE_EXTENDED : TPCANMessageType.
PCAN_MESSAGE_STANDARD;
            // If a remote frame will be sent, the data bytes are not important.
            //
            if (chbRemote.Checked)
                CANMsg.MSGTYPE |= TPCANMessageType.PCAN_MESSAGE_RTR;
            else
            {
                // We get so much data as the Len of the message
                //
                txtbCurrentTextBox = txtData0;
                for (int i = 0; i < CANMsg.LEN; i++)
                {
                    CANMsg.DATA[i] = Convert.ToByte(txtbCurrentTextBox.Text, 16);
                    if (i < 7)
                        txtbCurrentTextBox = (TextBox)this.GetNextControl(txtbCurrentTextBox, true);
                }
            }

            // The message is sent to the configured hardware
            //
            stsResult = PCANBasic.Write(m_PcanHandle, ref CANMsg);

            // The message was successfully sent
            //
            if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                IncludeTextMessage("Message was successfully SENT");
            // An error occurred.  We show the error.
            //
            else
                MessageBox.Show(GetFormatedError(stsResult));
        }

        private void btnReset_Click(object sender, EventArgs e)
        {
          TPCANStatus stsResult;

          // Resets the receive and transmit queues of a PCAN Channel.
          //
          stsResult = PCANBasic.Reset(m_PcanHandle);

          // If it fails, a error message is shown
          //
          if (stsResult != TPCANStatus.PCAN_ERROR_OK)
             MessageBox.Show(GetFormatedError(stsResult));
          else
             IncludeTextMessage("Receive and transmit queues successfully reset");
        }

        private void btnStatus_Click(object sender, EventArgs e)
        {
           TPCANStatus stsResult;
           String errorName;

           // Gets the current BUS status of a PCAN Channel.
           //
           stsResult = PCANBasic.GetStatus(m_PcanHandle);

           // Switch On Error Name
           //
           switch(stsResult)
           {
              case TPCANStatus.PCAN_ERROR_INITIALIZE:
                 errorName = "PCAN_ERROR_INITIALIZE";
                 break;
```

```
            case TPCANStatus.PCAN_ERROR_BUSLIGHT:
                errorName = "PCAN_ERROR_BUSLIGHT";
                break;

            case TPCANStatus.PCAN_ERROR_BUSHEAVY:
                errorName = "PCAN_ERROR_BUSHEAVY";
                break;

            case TPCANStatus.PCAN_ERROR_BUSOFF:
                errorName = "PCAN_ERROR_BUSOFF";
                break;

            case TPCANStatus.PCAN_ERROR_OK:
                errorName = "PCAN_ERROR_OK";
                break;

            default:
                errorName = "See Documentation";
                break;
        }

        // Display Message
        //
        IncludeTextMessage(String.Format("Status: {0} ({1:X}h)", errorName, stsResult));
    }
    #endregion

    #region Timer event-handler
    private void tmrRead_Tick(object sender, EventArgs e)
    {
        // Checks if in the receive-queue are currently messages for read
        //
        ReadMessages();
    }

    private void tmrDisplay_Tick(object sender, EventArgs e)
    {
        DisplayMessages();
    }
    #endregion

    #region Message List-View event-handler
    private void lstMessages_DoubleClick(object sender, EventArgs e)
    {
        // Clears the content of the Message List-View
        //
        btnMsgClear_Click(this, new EventArgs());
    }
    #endregion

    #region Information List-Box event-handler
    private void lbxInfo_DoubleClick(object sender, EventArgs e)
    {
        // Clears the content of the Information List-Box
        //
        btnInfoClear_Click(this, new EventArgs());
    }
    #endregion

    #region Textbox event handlers
    private void txtID_Leave(object sender, EventArgs e)
    {
        int iTextLength;
        uint uiMaxValue;

        // Calculates the text length and Maximum ID value according
        // with the Message Type
        //
        iTextLength = (chbExtended.Checked) ? 8 : 3;
        uiMaxValue = (chbExtended.Checked) ? (uint)0x1FFFFFF : (uint)0x7FF;

        // The Textbox for the ID is represented with 3 characters for
        // Standard and 8 characters for extended messages.
        // Therefore if the Length of the text is smaller than TextLength,
        // we add "0"
        //
```

```
        while (txtID.Text.Length != iTextLength)
            txtID.Text = ("0" + txtID.Text);

        // We check that the ID is not bigger than current maximum value
        //
        if (Convert.ToUInt32(txtID.Text, 16) > uiMaxValue)
            txtID.Text = string.Format("{0:X" + iTextLength.ToString() + "}", uiMaxValue);
    }

    private void txtID_KeyPress(object sender, KeyPressEventArgs e)
    {
        char chCheck;

        // We convert the Character to its Upper case equivalent
        //
        chCheck = char.ToUpper(e.KeyChar);

        // The Key is the Delete (Backspace) Key
        //
        if (chCheck == 8)
            return;
        // The Key is a number between 0-9
        //
        if ((chCheck > 47) && (chCheck < 58))
            return;
        // The Key is a character between A-F
        //
        if ((chCheck > 64) && (chCheck < 71))
            return;

        // Is neither a number nor a character between A(a) and F(f)
        //
        e.Handled = true;
    }

    private void txtData0_Leave(object sender, EventArgs e)
    {
        TextBox txtbCurrentTextbox;

        // all the Textbox Data fields are represented with 2 characters.
        // Therefore if the Length of the text is smaller than 2, we add
        // a "0"
        //
        if (sender.GetType().Name == "TextBox")
        {
            txtbCurrentTextbox = (TextBox)sender;
            while (txtbCurrentTextbox.Text.Length != 2)
                txtbCurrentTextbox.Text = ("0" + txtbCurrentTextbox.Text);
        }
    }
    #endregion

    #region Radio- and Check- Buttons event-handlers
    private void chbShowPeriod_CheckedChanged(object sender, EventArgs e)
    {
        // According with the check-value of this checkbox,
        // the recieved time of a messages will be interpreted as
        // period (time between the two last messages) or as time-stamp
        // (the elapsed time since windows was started)
        //
        lock (m_LastMsgsList.SyncRoot)
        {
            foreach (MessageStatus msg in m_LastMsgsList)
                msg.ShowingPeriod = chbShowPeriod.Checked;
        }
    }

    private void chbExtended_CheckedChanged(object sender, EventArgs e)
    {
        uint uiTemp;

        txtID.MaxLength = (chbExtended.Checked) ? 8 : 3;

        // the only way that the text length can be bigger als MaxLength
        // is when the change is from Extended to Standard message Type.
        // We have to handle this and set an ID not bigger than the Maximum
```

```
        // ID value for a Standard Message (0x7FF)
        //
        if (txtID.Text.Length > txtID.MaxLength)
        {
            uiTemp = Convert.ToUInt32(txtID.Text, 16);
            txtID.Text = (uiTemp < 0x7FF) ? string.Format("{0:X3}", uiTemp) : "7FF";
        }

        txtID_Leave(this, new EventArgs());
    }

    private void chbRemote_CheckedChanged(object sender, EventArgs e)
    {
        TextBox txtbCurrentTextBox;

        txtbCurrentTextBox = txtData0;

        // If the message is a RTR, no data is sent. The textbox for data
        // will be turned invisible
        //
        for (int i = 0; i < 8; i++)
        {
            txtbCurrentTextBox.Visible = !chbRemote.Checked;
            if (i < 7)
                txtbCurrentTextBox = (TextBox)this.GetNextControl(txtbCurrentTextBox, true);
        }
    }

    private void chbFilterExt_CheckedChanged(object sender, EventArgs e)
    {
        int iMaxValue;

        iMaxValue  = (chbFilterExt.Checked) ? 0x1FFFFFFF : 0x7FF;

        // We check that the maximum value for a selected filter
        // mode is used
        //
        if (nudIdTo.Value > iMaxValue)
            nudIdTo.Value = iMaxValue;

        nudIdTo.Maximum = iMaxValue;
        if (nudIdFrom.Value > iMaxValue)
            nudIdFrom.Value = iMaxValue;

        nudIdFrom.Maximum = iMaxValue;
    }

    private void rdbTimer_CheckedChanged(object sender, EventArgs e)
    {
        if (!btnRelease.Enabled)
            return;

        // According with the kind of reading, a timer, a thread or a button will be enabled
        //
        if (rdbTimer.Checked)
        {
            // Abort Read Thread if it exists
            //
            if (m_ReadThread != null)
            {
                m_ReadThread.Abort();
                m_ReadThread.Join();
                m_ReadThread = null;
            }

            // Enable Timer
            //
            tmrRead.Enabled = btnRelease.Enabled;
        }
        if (rdbEvent.Checked)
        {
            // Disable Timer
            //
            tmrRead.Enabled = false;
            // Create and start the tread to read CAN Message using SetRcvEvent()
            //
```

```
                System.Threading.ThreadStart threadDelegate = new System.Threading.ThreadStart(this.
    CANReadThreadFunc);
                m_ReadThread = new System.Threading.Thread(threadDelegate);
                m_ReadThread.IsBackground = true;
                m_ReadThread.Start();
            }
            if (rdbManual.Checked)
            {
                // Abort Read Thread if it exists
                //
                if (m_ReadThread != null)
                {
                    m_ReadThread.Abort();
                    m_ReadThread.Join();
                    m_ReadThread = null;
                }
                // Disable Timer
                //
                tmrRead.Enabled = false;
            }
            btnRead.Enabled = btnRelease.Enabled && rdbManual.Checked;
        }
        #endregion

        private void saveAsToolStripMenuItem_Click(object sender, EventArgs e)
        {
            Application.Restart();
        }

        #endregion

        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }

        #endregion

        private void helpToolStripMenuItem_Click(object sender, EventArgs e)
        {

        }
    }
}
```

10 ANNEXE 2 : Code source du programme pour PC

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;


namespace Calcuatrice
{
    public partial class Calculatrice : Form
    {

        string nombre, nombre2, TypeDeCalcul, EqualIsPressed;
        double Result, stack;


        public Calculatrice()
        {
            InitializeComponent();
            nombre2 = "0";
            nombre = "0";
            EqualIsPressed = "0";
            stack = 0;
        }

        private void btn1_Click(object sender, EventArgs e)
        {
            if (EqualIsPressed == "1")
            {
                EqualIsPressed = "0";
                textBox.Text = "";
                label.Text = "";
                nombre = "0";
                nombre2 = "0";
            }


            textBox.Text = textBox.Text + "1";
            nombre = textBox.Text;
            label.Text = label.Text + "1";
        }

        private void btn2_Click(object sender, EventArgs e)
        {
            if (EqualIsPressed == "1")
            {
                EqualIsPressed = "0";
                textBox.Text = "";
                label.Text = "";
                nombre = "0";
                nombre2 = "0";
            }
            textBox.Text = textBox.Text + "2";
            nombre = textBox.Text;
            label.Text = label.Text + "2";
        }

        private void btn3_Click(object sender, EventArgs e)
        {
            if (EqualIsPressed == "1")
            {
                EqualIsPressed = "0";
                textBox.Text = "";
                label.Text = "";
                nombre = "0";
                nombre2 = "0";
            }
            textBox.Text = textBox.Text + "3";
            nombre = textBox.Text;
            label.Text = label.Text + "3";
```

```
    }

    private void btn4_Click(object sender, EventArgs e)
    {
        if (EqualIsPressed == "1")
        {
            EqualIsPressed = "0";
            textBox.Text = "";
            label.Text = "";
            nombre = "0";
            nombre2 = "0";
        }
        textBox.Text = textBox.Text + "4";
        nombre = textBox.Text;
        label.Text = label.Text + "4";
    }

    private void btn5_Click(object sender, EventArgs e)
    {
        if (EqualIsPressed == "1")
        {
            EqualIsPressed = "0";
            textBox.Text = "";
            label.Text = "";
            nombre = "0";
            nombre2 = "0";
        }
        textBox.Text = textBox.Text + "5";
        nombre = textBox.Text;
        label.Text = label.Text + "5";
    }

    private void btn6_Click(object sender, EventArgs e)
    {
        if (EqualIsPressed == "1")
        {
            EqualIsPressed = "0";
            textBox.Text = "";
            label.Text = "";
            nombre = "0";
            nombre2 = "0";
        }
        textBox.Text = textBox.Text + "6";
        nombre = textBox.Text;
        label.Text = label.Text + "6";
    }

    private void btn7_Click(object sender, EventArgs e)
    {
        if (EqualIsPressed == "1")
        {
            EqualIsPressed = "0";
            textBox.Text = "";
            label.Text = "";
            nombre = "0";
            nombre2 = "0";
        }
        textBox.Text = textBox.Text + "7";
        nombre = textBox.Text;
        label.Text = label.Text + "7";
    }

    private void btn8_Click(object sender, EventArgs e)
    {
        if (EqualIsPressed == "1")
        {
            EqualIsPressed = "0";
            textBox.Text = "";
            label.Text = "";
            nombre = "0";
            nombre2 = "0";
        }
        textBox.Text = textBox.Text + "8";
        nombre = textBox.Text;
        label.Text = label.Text + "8";
    }
```

```
private void btn9_Click(object sender, EventArgs e)
{
    if (EqualIsPressed == "1")
    {
        EqualIsPressed = "0";
        textBox.Text = "";
        label.Text = "";
        nombre = "0";
        nombre2 = "0";
    }
    textBox.Text = textBox.Text + "9";
    nombre = textBox.Text;
    label.Text = label.Text + "9";
}

private void btn0_Click(object sender, EventArgs e)
{
    if (EqualIsPressed == "1")
    {
        EqualIsPressed = "0";
        textBox.Text = "";
        label.Text = "";
        nombre = "0";
        nombre2 = "0";
    }
    textBox.Text = textBox.Text + "0";
    nombre = textBox.Text;
    label.Text = label.Text + "0";
}

private void btn_Clear_Click(object sender, EventArgs e)
{
    textBox.Text = "";
    label.Text = "";
    nombre = "0";
    nombre2 = "0";
}




private void btn_virg_Click(object sender, EventArgs e)
{
    if (EqualIsPressed == "1")
    {
        EqualIsPressed = "0";
        textBox.Text = "";
        label.Text = "";
        nombre = "0";
        nombre2 = "0";
    }

    if (textBox.Text.Contains(","))
    {
        return;
    }

    textBox.Text = textBox.Text + ",";
    nombre = textBox.Text;
    label.Text = label.Text + ",";
}




private void btn_moins_Click(object sender, EventArgs e)
{
    if (textBox.Text == "")
    {
        return;
```

```
        }
        label.Text = label.Text + " - ";
        textBox.Text = "";
        TypeDeCalcul = "-";
        nombre2 = nombre;
    }




    private void btnx_Click(object sender, EventArgs e)
    {
        if (textBox.Text == "")
        {
            return;
        }
        label.Text = label.Text + " x ";
        textBox.Text = "";
        TypeDeCalcul = "x";
        nombre2 = nombre;
    }

    private void btn_divide_Click(object sender, EventArgs e)
    {
        if (textBox.Text == "")
        {
            return;
        }
        label.Text = label.Text + " / ";
        textBox.Text = "";
        TypeDeCalcul = "/";
        nombre2 = nombre;
    }

    private void btn_Sqrt_Click(object sender, EventArgs e)
    {
        if (textBox.Text == "")
        {
            return;
        }
        label.Text = " sqrt (" + label.Text + ")" ;
        textBox.Text = "";
        nombre2 = nombre;

        Result = Math.Sqrt(Convert.ToDouble(nombre2));
        textBox.Text = Convert.ToString(Result);

        label.Text = label.Text + " = ";
        label.Text = label.Text + textBox.Text;
        EqualIsPressed = "1";
    }

    private void btn_surX_Click(object sender, EventArgs e)
    {
        if (textBox.Text == "")
        {
            return;
        }
        else
        {
            label.Text = "1 / " + label.Text;
            textBox.Text = "1/" + textBox.Text;
            TypeDeCalcul = "1/x";
            nombre2 = nombre;
        }
    }

    private void btn_plus_Click(object sender, EventArgs e)
    {
        if(textBox.Text == "")
        {
            return;
        }
        label.Text = label.Text + " + ";
        textBox.Text = "";
```

```
            TypeDeCalcul = "+";
            nombre2 = nombre;
            stack = stack + Convert.ToDouble(nombre2);
        }




        private void btn_Equal_Click(object sender, EventArgs e)
        {
            EqualIsPressed = "1";

            label.Text = label.Text + " = ";

            switch (TypeDeCalcul)
            {
                case("+"):
                    Result = Convert.ToDouble(nombre2) + Convert.ToDouble(nombre);
                    textBox.Text = Convert.ToString(Result);
                break;

                case ("-"):
                    Result = Convert.ToDouble(nombre2) - Convert.ToDouble(nombre);
                    textBox.Text = Convert.ToString(Result);
                break;

                case ("x"):
                    Result = Convert.ToDouble(nombre2) * Convert.ToDouble(nombre);
                    textBox.Text = Convert.ToString(Result);
                break;

                case ("/"):

                if (Convert.ToDouble(nombre) == 0)
                {
                    textBox.Text = "You created a Dark Hole!";
                    Form2 frm = new Form2();
                    frm.Show();
                    return;
                }
                    Result = Convert.ToDouble(nombre2) / Convert.ToDouble(nombre);
                    textBox.Text = Convert.ToString(Result);
                break;

                case ("1/x"):
                    Result = 1 / Convert.ToDouble(nombre2);
                    textBox.Text = Convert.ToString(Result);
                break;
            }
            label.Text = label.Text + textBox.Text;
        }

    }
}
```

11 ANNEXE 6 : Code source du programme PIC

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

/// <summary>
/// Inclusion of PEAK PCAN-Basic namespace
/// </summary>
using Peak.Can.Basic;
using TPCANHandle = System.Byte;

namespace ICDIBasic
{
    public partial class Form1 : Form
    {
        #region Structures
        /// <summary>
        /// Message Status structure used to show CAN Messages
        /// in a ListView
        /// </summary>
        private class MessageStatus
        {
            private TPCANMsg m_Msg;
            private TPCANTimestamp m_TimeStamp;
            private TPCANTimestamp m_oldTimeStamp;
            private int m_iIndex;
            private int m_Count;
            private bool m_bShowPeriod;
            private bool m_bWasChanged;

            public MessageStatus(TPCANMsg canMsg, TPCANTimestamp canTimestamp, int listIndex)
            {
                m_Msg = canMsg;
                m_TimeStamp = canTimestamp;
                m_oldTimeStamp = canTimestamp;
                m_iIndex = listIndex;
                m_Count = 1;
                m_bShowPeriod = true;
                m_bWasChanged = false;
            }

            public void Update(TPCANMsg canMsg, TPCANTimestamp canTimestamp)
            {
                m_Msg = canMsg;
                m_oldTimeStamp = m_TimeStamp;
                m_TimeStamp = canTimestamp;
                m_bWasChanged = true;
                m_Count += 1;
            }

            public TPCANMsg CANMsg
            {
                get { return m_Msg; }
            }

            public TPCANTimestamp Timestamp
            {
                get { return m_TimeStamp; }
            }

            public int Position
            {
                get { return m_iIndex; }
            }

            public string TypeString
            {
                get { return GetMsgTypeString(); }
            }

            public string IdString
```

```
        {
            get { return GetIdString(); }
        }

        public string DataString
        {
            get { return GetDataString(); }
        }

        public int Count
        {
            get { return m_Count; }
        }

        public bool ShowingPeriod
        {
            get { return m_bShowPeriod; }
            set
            {
                if (m_bShowPeriod ^ value)
                {
                    m_bShowPeriod = value;
                    m_bWasChanged = true;
                }
            }
        }

        public bool MarkedAsUpdated
        {
            get { return m_bWasChanged; }
            set { m_bWasChanged = value; }
        }

        public string TimeString
        {
            get { return GetTimeString(); }
        }

        private string GetTimeString()
        {
            double fTime;

            fTime = m_TimeStamp.millis + (m_TimeStamp.micros / 1000.0);
            if (m_bShowPeriod)
                fTime -= (m_oldTimeStamp.millis + (m_oldTimeStamp.micros / 1000.0));
            return fTime.ToString("F1");
        }

        private string GetDataString()
        {
            string strTemp;

            strTemp = "";

            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_RTR) == TPCANMessageType.PCAN_MESSAGE_RTR)
                return "Remote Request";
            else
                for (int i = 0; i < m_Msg.LEN; i++)
                    strTemp += string.Format("{0:X2} ", m_Msg.DATA[i]);

            return strTemp;
        }

        private string GetIdString()
        {
            // We format the ID of the message and show it
            //
            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_EXTENDED) == TPCANMessageType.
PCAN_MESSAGE_EXTENDED)
                return string.Format("{0:X8}h", m_Msg.ID);
            else
                return string.Format("{0:X3}h", m_Msg.ID);
        }

        private string GetMsgTypeString()
        {
```

```
            string strTemp;

            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_EXTENDED) == TPCANMessageType.
PCAN_MESSAGE_EXTENDED)
                strTemp = "EXTENDED";
            else
                strTemp = "STANDARD";

            if ((m_Msg.MSGTYPE & TPCANMessageType.PCAN_MESSAGE_RTR) == TPCANMessageType.PCAN_MESSAGE_RTR)
                strTemp += "/RTR";

            return strTemp;
        }

    }
    #endregion

    #region Delegates
    /// <summary>
    /// Read-Delegate Handler
    /// </summary>
    private delegate void ReadDelegateHandler();
    #endregion

    #region Members
    /// <summary>
    /// Saves the handle of a PCAN hardware
    /// </summary>
    private TPCANHandle m_PcanHandle;
    /// <summary>
    /// Saves the baudrate register for a conenction
    /// </summary>
    private TPCANBaudrate m_Baudrate;
    /// <summary>
    /// Saves the type of a non-plug-and-play hardware
    /// </summary>
    private TPCANType m_HwType;
    /// <summary>
    /// Stores the status of received messages for its display
    /// </summary>
    private System.Collections.ArrayList m_LastMsgsList;
    /// <summary>
    /// Read Delegate for calling the function "ReadMessages"
    /// </summary>
    private ReadDelegateHandler m_ReadDelegate;
    /// <summary>
    /// Receive-Event
    /// </summary>
    private System.Threading.AutoResetEvent m_ReceiveEvent;
    /// <summary>
    /// Thread for message reading (using events)
    /// </summary>
    private System.Threading.Thread m_ReadThread;
    /// <summary>
    /// Handles of the current available PCAN-Hardware
    /// </summary>
    private TPCANHandle[] m_HandlesArray;
    #endregion

    #region Methods
    #region Help functions
    /// <summary>
    /// Initialization of PCAN-Basic components
    /// </summary>
    private void InitializeBasicComponents()
    {
        // Creates the list for received messages
        //
        m_LastMsgsList = new System.Collections.ArrayList();
        // Creates the delegate used for message reading
        //
        m_ReadDelegate = new ReadDelegateHandler(ReadMessages);
        // Creates the event used for signalize incomming messages
        //
        m_ReceiveEvent = new System.Threading.AutoResetEvent(false);
        // Creates an array with all possible PCAN-Channels
```

```
            //
            m_HandlesArray = new TPCANHandle[]
            {
                PCANBasic.PCAN_ISABUS1,
                PCANBasic.PCAN_ISABUS2,
                PCANBasic.PCAN_ISABUS3,
                PCANBasic.PCAN_ISABUS4,
                PCANBasic.PCAN_ISABUS5,
                PCANBasic.PCAN_ISABUS6,
                PCANBasic.PCAN_ISABUS7,
                PCANBasic.PCAN_ISABUS8,
                PCANBasic.PCAN_DNGBUS1,
                PCANBasic.PCAN_PCIBUS1,
                PCANBasic.PCAN_PCIBUS2,
                PCANBasic.PCAN_PCIBUS3,
                PCANBasic.PCAN_PCIBUS4,
                PCANBasic.PCAN_PCIBUS5,
                PCANBasic.PCAN_PCIBUS6,
                PCANBasic.PCAN_PCIBUS7,
                PCANBasic.PCAN_PCIBUS8,
                PCANBasic.PCAN_USBBUS1,
                PCANBasic.PCAN_USBBUS2,
                PCANBasic.PCAN_USBBUS3,
                PCANBasic.PCAN_USBBUS4,
                PCANBasic.PCAN_USBBUS5,
                PCANBasic.PCAN_USBBUS6,
                PCANBasic.PCAN_USBBUS7,
                PCANBasic.PCAN_USBBUS8,
                PCANBasic.PCAN_PCCBUS1,
                PCANBasic.PCAN_PCCBUS2
            };

            // Fills and configures the Data of several comboBox components
            //
            FillComboBoxData();

            // Prepares the PCAN-Basic's debug-Log file
            //
            ConfigureLogFile();
        }

        /// <summary>
        /// COnfigures the Debug-Log file of PCAN-Basic
        /// </summary>
        private void ConfigureLogFile()
        {
            UInt32 iBuffer;

            // Sets the mask to catch all events
            //
            iBuffer = PCANBasic.LOG_FUNCTION_ENTRY | PCANBasic.LOG_FUNCTION_LEAVE | PCANBasic.
LOG_FUNCTION_PARAMETERS |
                PCANBasic.LOG_FUNCTION_READ | PCANBasic.LOG_FUNCTION_WRITE;

            // Configures the log file.
            // NOTE: The Log capability is to be used with the NONEBUS Handle. Other handle than this will
            // cause the function fail.
            //
            PCANBasic.SetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_LOG_CONFIGURE, ref iBuffer, sizeof(
UInt32));
        }

        /// <summary>
        /// Help Function used to get an error as text
        /// </summary>
        /// <param name="error">Error code to be translated</param>
        /// <returns>A text with the translated error</returns>
        private string GetFormatedError(TPCANStatus error)
        {
            StringBuilder strTemp;

            // Creates a buffer big enough for a error-text
            //
            strTemp = new StringBuilder(256);
            // Gets the text using the GetErrorText API function
            // If the function success, the translated error is returned. If it fails,
```

```
            // a text describing the current error is returned.
            //
            if (PCANBasic.GetErrorText(error, 0, strTemp) != TPCANStatus.PCAN_ERROR_OK)
                return string.Format("An error occurred. Error-code's text ({0:X}) couldn't be retrieved", error
);
            else
                return strTemp.ToString();
        }

        /// <summary>
        /// Includes a new line of text into the information Listview
        /// </summary>
        /// <param name="strMsg">Text to be included</param>
        private void IncludeTextMessage(string strMsg)
        {
            lbxInfo.Items.Add(strMsg);
            lbxInfo.SelectedIndex = lbxInfo.Items.Count - 1;
        }

        /// <summary>
        /// Gets the current status of the PCAN-Basic message filter
        /// </summary>
        /// <param name="status">Buffer to retrieve the filter status</param>
        /// <returns>If calling the function was successfull or not</returns>
        private bool GetFilterStatus(out uint status)
        {
            TPCANStatus stsResult;

            // Tries to get the sttaus of the filter for the current connected hardware
            //
            stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_MESSAGE_FILTER, out status, sizeof(
UInt32));

            // If it fails, a error message is shown
            //
            if (stsResult != TPCANStatus.PCAN_ERROR_OK)
            {
                MessageBox.Show(GetFormatedError(stsResult));
                return false;
            }
            return true;
        }

        /// <summary>
        /// Configures the data of all ComboBox components of the main-form
        /// </summary>
        private void FillComboBoxData()
        {
            // Channels will be check
            //
            btnHwRefresh_Click(this, new EventArgs());

            // Baudrates
            //
            cbbBaudrates.SelectedIndex = 2; // 500 K

            // Hardware Type for no plugAndplay hardware
            //
            cbbHwType.SelectedIndex = 0;

            // Interrupt for no plugAndplay hardware
            //
            cbbInterrupt.SelectedIndex = 0;

            // IO Port for no plugAndplay hardware
            //
            cbbIO.SelectedIndex = 0;

            // Parameters for GetValue and SetValue function calls
            //
            cbbParameter.SelectedIndex = 0;
        }

        /// <summary>
        /// Activates/deaactivates the different controls of the main-form according
        /// with the current connection status
```

```
        /// </summary>
        /// <param name="bConnected">Current status. True if connected, false otherwise</param>
        private void SetConnectionStatus(bool bConnected)
        {
            // Buttons
            //
            btnInit.Enabled = !bConnected;
            btnRead.Enabled = bConnected && rdbManual.Checked;
            btnWrite.Enabled = bConnected;
            btnRelease.Enabled = bConnected;
            btnFilterApply.Enabled = bConnected;
            btnFilterQuery.Enabled = bConnected;
            btnParameterSet.Enabled = bConnected;
            btnParameterGet.Enabled = bConnected;
            btnGetVersions.Enabled = bConnected;
            btnHwRefresh.Enabled = !bConnected;
            btnStatus.Enabled = bConnected;
            btnReset.Enabled = bConnected;

            // ComboBoxs
            //
            cbbBaudrates.Enabled = !bConnected;
            cbbChannel.Enabled = !bConnected;
            cbbHwType.Enabled = !bConnected;
            cbbIO.Enabled = !bConnected;
            cbbInterrupt.Enabled = !bConnected;

            // Hardware configuration and read mode
            //
            if (!bConnected)
                cbbChannel_SelectedIndexChanged(this, new EventArgs());
            else
                rdbTimer_CheckedChanged(this, new EventArgs());

            // Display messages in grid
            //
            tmrDisplay.Enabled = bConnected;
        }

        /// <summary>
        /// Gets the formated text for a CPAN-Basic channel handle
        /// </summary>
        /// <param name="handle">PCAN-Basic Handle to format</param>
        /// <returns>The formatted text for a channel</returns>
        private string FormatChannelName(TPCANHandle handle)
        {
            TPCANDevice devDevice;
            byte byChannel;

            // Gets the owner device and channel for a
            // PCAN-Basic handle
            //
            devDevice = (TPCANDevice)(handle >> 4);
            byChannel = (byte)(handle & 0xF);

            // Constructs the PCAN-Basic Channel name and return it
            //
            return string.Format("{0} {1} ({2:X2}h)", devDevice, byChannel, handle);
        }
        #endregion

        #region Message-proccessing functions
        /// <summary>
        /// Display CAN messages in the Message-ListView
        /// </summary>
        private void DisplayMessages()
        {
            ListViewItem lviCurrentItem;

            lock (m_LastMsgsList.SyncRoot)
            {
                foreach (MessageStatus msgStatus in m_LastMsgsList)
                {
                    // Get the data to actualize
                    //
                    if (msgStatus.MarkedAsUpdated)
```

```
            {
                msgStatus.MarkedAsUpdated = false;
                lviCurrentItem = lstMessages.Items[msgStatus.Position];

                lviCurrentItem.SubItems[2].Text = msgStatus.CANMsg.LEN.ToString();
                lviCurrentItem.SubItems[3].Text = msgStatus.DataString;
                lviCurrentItem.SubItems[4].Text = msgStatus.Count.ToString();
                lviCurrentItem.SubItems[5].Text = msgStatus.TimeString;
            }
        }
    }
}

/// <summary>
/// Inserts a new entry for a new message in the Message-ListView
/// </summary>
/// <param name="newMsg">The messasge to be inserted</param>
/// <param name="timeStamp">The Timesamp of the new message</param>
private void InsertMsgEntry(TPCANMsg newMsg, TPCANTimestamp timeStamp)
{
    MessageStatus msgStsCurrentMsg;
    ListViewItem lviCurrentItem;

    lock (m_LastMsgsList.SyncRoot)
    {
        // We add this status in the last message list
        //
        msgStsCurrentMsg = new MessageStatus(newMsg, timeStamp, lstMessages.Items.Count);
        m_LastMsgsList.Add(msgStsCurrentMsg);

        // Add the new ListView Item with the Type of the message
        //
        lviCurrentItem = lstMessages.Items.Add(msgStsCurrentMsg.TypeString);
        // We set the ID of the message
        //
        lviCurrentItem.SubItems.Add(msgStsCurrentMsg.IdString);
        // We set the length of the Message
        //
        lviCurrentItem.SubItems.Add(newMsg.LEN.ToString());
        // We set the data of the message.
        //
        lviCurrentItem.SubItems.Add(msgStsCurrentMsg.DataString);
        // we set the message count message (this is the First, so count is 1)
        //
        lviCurrentItem.SubItems.Add(msgStsCurrentMsg.Count.ToString());
        // Add time stamp information if needed
        //
        lviCurrentItem.SubItems.Add(msgStsCurrentMsg.TimeString);
    }
}

/// <summary>
/// Processes a received message, in order to show it in the Message-ListView
/// </summary>
/// <param name="theMsg">The received PCAN-Basic message</param>
/// <returns>True if the message must be created, false if it must be modified</returns>
private void ProcessMessage(TPCANMsg theMsg, TPCANTimestamp itsTimeStamp)
{
    // We search if a message (Same ID and Type) is
    // already received or if this is a new message
    //
    lock (m_LastMsgsList.SyncRoot)
    {
        foreach (MessageStatus msg in m_LastMsgsList)
        {
            if ((msg.CANMsg.ID == theMsg.ID) && (msg.CANMsg.MSGTYPE == theMsg.MSGTYPE))
            {
                // Modify the message and exit
                //
                msg.Update(theMsg, itsTimeStamp);
                return;
            }
        }
        // Message not found. It will created
        //
        InsertMsgEntry(theMsg, itsTimeStamp);
```

```
            }
        }

        /// <summary>
        /// Thread-Function used for reading PCAN-Basic messages
        /// </summary>
        private void CANReadThreadFunc()
        {
            UInt32 iBuffer;
            TPCANStatus stsResult;

            iBuffer = Convert.ToUInt32(m_ReceiveEvent.SafeWaitHandle.DangerousGetHandle().ToInt32());
            // Sets the handle of the Receive-Event.
            //
            stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_RECEIVE_EVENT, ref iBuffer, sizeof(
UInt32));

            if(stsResult != TPCANStatus.PCAN_ERROR_OK)
          {
             MessageBox.Show(GetFormatedError(stsResult),"Error!",MessageBoxButtons.OK,MessageBoxIcon.Error);
             return;
          }

            // While this mode is selected
            while (rdbEvent.Checked)
            {
                // Waiting for Receive-Event
                //
                if (m_ReceiveEvent.WaitOne(50))
                    // Process Receive-Event using .NET Invoke function
                    // in order to interact with Winforms UI (calling the
                    // function ReadMessages)
                    //
                    this.Invoke(m_ReadDelegate);
            }
        }

        /// <summary>
        /// Function for reading PCAN-Basic messages
        /// </summary>
        private void ReadMessages()
        {
            TPCANMsg CANMsg;
            TPCANTimestamp CANTimeStamp;
            TPCANStatus stsResult;

            // We read at least one time the queue looking for messages.
            // If a message is found, we look again trying to find more.
            // If the queue is empty or an error occurr, we get out from
            // the dowhile statement.
            //
            do
            {
                // We execute the "Read" function of the PCANBasic
                //
                stsResult = PCANBasic.Read(m_PcanHandle, out CANMsg, out CANTimeStamp);

                // A message was received
                // We process the message(s)
                //
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    ProcessMessage(CANMsg, CANTimeStamp);

            } while (btnRelease.Enabled && (!Convert.ToBoolean(stsResult & TPCANStatus.PCAN_ERROR_QRCVEMPTY)));
        }
        #endregion

        #region Event Handlers
        #region Form event-handlers
        /// <summary>
        /// Consturctor
        /// </summary>
        public Form1()
        {
            // Initializes Form's component
            //
```

```
        InitializeComponent();
        // Initializes specific components
        //
        InitializeBasicComponents();
    }

    /// <summary>
    /// Form-Closing Function / Finish function
    /// </summary>
    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
        // Releases the used PCAN-Basic channel
        //
        PCANBasic.Uninitialize(m_PcanHandle);
    }
    #endregion

    #region ComboBox event-handlers
    private void cbbChannel_SelectedIndexChanged(object sender, EventArgs e)
    {
        bool bNonPnP;
        string strTemp;

        // Get the handle fromt he text being shown
        //
        strTemp = cbbChannel.Text;
        strTemp = strTemp.Substring(strTemp.IndexOf('(')+1, 2);

        // Determines if the handle belong to a No Plug&Play hardware
        //
        m_PcanHandle = Convert.ToByte(strTemp,16);
        bNonPnP = m_PcanHandle <= PCANBasic.PCAN_DNGBUS1;
        // Activates/deactivates configuration controls according with the
        // kind of hardware
        //
        cbbHwType.Enabled = bNonPnP;
        cbbIO.Enabled = bNonPnP;
        cbbInterrupt.Enabled = bNonPnP;
    }

    private void cbbBaudrates_SelectedIndexChanged(object sender, EventArgs e)
    {
        // Saves the current selected baudrate register code
        //
        switch (cbbBaudrates.SelectedIndex)
        {
            case 0:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_1M;
                break;
            case 1:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_800K;
                break;
            case 2:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_500K;
                break;
            case 3:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_250K;
                break;
            case 4:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_125K;
                break;
            case 5:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_100K;
                break;
            case 6:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_95K;
                break;
            case 7:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_83K;
                break;
            case 8:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_50K;
                break;
            case 9:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_47K;
                break;
```

```
            case 10:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_33K;
                break;
            case 11:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_20K;
                break;
            case 12:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_10K;
                break;
            case 13:
                m_Baudrate = TPCANBaudrate.PCAN_BAUD_5K;
                break;
        }
    }

    private void cbbHwType_SelectedIndexChanged(object sender, EventArgs e)
    {
        // Saves the current type for a no-Plug&Play hardware
        //
        switch (cbbHwType.SelectedIndex)
        {
            case 0:
                m_HwType = TPCANType.PCAN_TYPE_ISA;
                break;
            case 1:
                m_HwType = TPCANType.PCAN_TYPE_ISA_SJA;
                break;
            case 2:
                m_HwType = TPCANType.PCAN_TYPE_ISA_PHYTEC;
                break;
            case 3:
                m_HwType = TPCANType.PCAN_TYPE_DNG;
                break;
            case 4:
                m_HwType = TPCANType.PCAN_TYPE_DNG_EPP;
                break;
            case 5:
                m_HwType = TPCANType.PCAN_TYPE_DNG_SJA;
                break;
            case 6:
                m_HwType = TPCANType.PCAN_TYPE_DNG_SJA_EPP;
                break;
        }
    }

    private void cbbParameter_SelectedIndexChanged(object sender, EventArgs e)
    {
        // Activates/deactivates controls according with the selected
        // PCAN-Basic parameter
        //
        rdbParamActive.Enabled = cbbParameter.SelectedIndex != 0;
        rdbParamInactive.Enabled = rdbParamActive.Enabled;
        nudDeviceId.Enabled = !rdbParamActive.Enabled;
    }
    #endregion

    #region Button event-handlers
    private void btnHwRefresh_Click(object sender, EventArgs e)
    {
        UInt32 iBuffer;
        TPCANStatus stsResult;

        // Clears the Channel combioBox and fill it again with
        // the PCAN-Basic handles for no-Plug&Play hardware and
        // the detected Plug&Play hardware
        //
        cbbChannel.Items.Clear();
        try
        {
            for (int i = 0; i < m_HandlesArray.Length; i++)
            {
                // Includes all no-Plug&Play Handles
                if (m_HandlesArray[i] <= PCANBasic.PCAN_DNGBUS1)
                    cbbChannel.Items.Add(FormatChannelName(m_HandlesArray[i]));
                else
                {
```

```
                    // Checks for a Plug&Play Handle and, according with the return value, includes it
                    // into the list of available hardware channels.
                    //
                    stsResult = PCANBasic.GetValue(m_HandlesArray[i], TPCANParameter.PCAN_CHANNEL_CONDITION,
 out iBuffer, sizeof(UInt32));
                    if ((stsResult == TPCANStatus.PCAN_ERROR_OK) && (iBuffer == PCANBasic.
PCAN_CHANNEL_AVAILABLE))
                        cbbChannel.Items.Add(FormatChannelName(m_HandlesArray[i]));
                }
            }
            cbbChannel.SelectedIndex = cbbChannel.Items.Count - 1;
        }
        catch (DllNotFoundException)
        {
            MessageBox.Show("Le fichier PCANBasic.dll est introuvable!", "Missing DLL Err0r!",
MessageBoxButtons.OK, MessageBoxIcon.Error);
            Environment.Exit(-1);
        }
    }

    private void btnInit_Click(object sender, EventArgs e)
    {
        TPCANStatus stsResult;

        // Connects a selected PCAN-Basic channel
        //
        stsResult = PCANBasic.Initialize(
            m_PcanHandle,
            m_Baudrate,
            m_HwType,
            Convert.ToUInt32(cbbIO.Text,16),
            Convert.ToUInt16(cbbInterrupt.Text));

        if (stsResult != TPCANStatus.PCAN_ERROR_OK)
            MessageBox.Show(GetFormatedError(stsResult));

        // Sets the connection status of the main-form
        //
        SetConnectionStatus(stsResult == TPCANStatus.PCAN_ERROR_OK);
    }

    private void btnRelease_Click(object sender, EventArgs e)
    {
        // Releases a current connected PCAN-Basic channel
        //
        PCANBasic.Uninitialize(m_PcanHandle);
        tmrRead.Enabled = false;
        if (m_ReadThread != null)
        {
            m_ReadThread.Abort();
            m_ReadThread.Join();
            m_ReadThread = null;
        }

        // Sets the connection status of the main-form
        //
        SetConnectionStatus(false);
    }

    private void btnFilterApply_Click(object sender, EventArgs e)
    {
        UInt32 iBuffer;
        TPCANStatus stsResult;

        // Gets the current status of the message filter
        //
        if (!GetFilterStatus(out iBuffer))
            return;

        // Configures the message filter for a custom range of messages
        //
        if (rdbFilterCustom.Checked)
        {
            // The filter must be first closed in order to customize it
            //
            if (iBuffer != PCANBasic.PCAN_FILTER_OPEN)
```

```
            {
                // Sets the custom filter
                //
                stsResult = PCANBasic.FilterMessages(
                m_PcanHandle,
                Convert.ToUInt32(nudIdFrom.Value),
                Convert.ToUInt32(nudIdTo.Value),
                chbFilterExt.Checked ? TPCANMode.PCAN_MODE_EXTENDED : TPCANMode.PCAN_MODE_STANDARD);
                // If success, an information message is written, if it is not, an error message is shown
                //
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The filter was customized. IDs from {0:X} to {1:X}
will be received", nudIdFrom.Text, nudIdTo.Text));
                else
                    MessageBox.Show(GetFormatedError(stsResult));
            }
            else
                MessageBox.Show("Le filtre doit être fermé si vous voulez le modifier");

            return;
        }

        // The filter will be full opened or complete closed
        //
        if (rdbFilterClose.Checked)
            iBuffer = PCANBasic.PCAN_FILTER_CLOSE;
        else
            iBuffer = PCANBasic.PCAN_FILTER_OPEN;

        // The filter is configured
        //
        stsResult = PCANBasic.SetValue(
            m_PcanHandle,
            TPCANParameter.PCAN_MESSAGE_FILTER,
            ref iBuffer,
            sizeof(UInt32));

        // If success, an information message is written, if it is not, an error message is shown
        //
        if (stsResult == TPCANStatus.PCAN_ERROR_OK)
           IncludeTextMessage(string.Format("The filter a été {0}", rdbFilterClose.Checked ? "closed." : "
opened."));
        else
            MessageBox.Show(GetFormatedError(stsResult));
    }

    private void btnFilterQuery_Click(object sender, EventArgs e)
    {
        UInt32 iBuffer;

        // Queries the current status of the message filter
        //
        if (GetFilterStatus(out iBuffer))
        {
            switch(iBuffer)
            {
                // The filter is closed
                //
                case  PCANBasic.PCAN_FILTER_CLOSE:
                    IncludeTextMessage("Le filtre est: fermé.");
                    break;
                // The filter is fully opened
                //
                case PCANBasic.PCAN_FILTER_OPEN:
                    IncludeTextMessage("Le filtre est: ouvert.");
                    break;
                // The filter is customized
                //
                case PCANBasic.PCAN_FILTER_CUSTOM:
                    IncludeTextMessage("Le filtre est: personnalisé.");
                    break;
                // The status of the filter is undefined. (Should never happen)
                //
                default:
                    IncludeTextMessage("Le filtre est: invalide.");
                    break;
```

```
            }
        }
    }

    private void btnParameterSet_Click(object sender, EventArgs e)
    {
        TPCANStatus stsResult;
        UInt32 iBuffer;

        // Sets a PCAN-Basic parameter value
        //
        switch (cbbParameter.SelectedIndex)
        {
            // The Device-Number of an USB channel will be set
            //
            case 0:
                iBuffer = Convert.ToUInt32(nudDeviceId.Value);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_DEVICE_NUMBER, ref iBuffer,
 sizeof(UInt32));
                if(stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage("The desired Device-Number was successfully configured");
                break;
            // The 5 Volt Power feature of a PC-card or USB will be set
            //
            case 1:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_5VOLTS_POWER, ref iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The USB/PC-Card 5 power was successfully {0}", (
iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The feature for automatic reset on BUS-OFF will be set
            //
            case 2:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_BUSOFF_AUTORESET, ref
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The automatic-reset on BUS-OFF was successfully {0}",
(iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The CAN option "Listen Only" will be set
            //
            case 3:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(m_PcanHandle, TPCANParameter.PCAN_LISTEN_ONLY, ref iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The CAN-option Listen-Only was successfully {0}", (
iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The feature for logging debug-information will be set
            //
            case 4:
                iBuffer = (uint)(rdbParamActive.Checked ? PCANBasic.PCAN_PARAMETER_ON : PCANBasic.
PCAN_PARAMETER_OFF);
                stsResult = PCANBasic.SetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_LOG_STATUS, ref
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The feature for logging debug information was
successfully {0}", (iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "activated" : "deactivated"));
                break;
            // The current parameter is invalid
            //
            default:
                stsResult = TPCANStatus.PCAN_ERROR_UNKNOWN;
                MessageBox.Show("Wrong parameter code.");
                return;
        }

        // If the function fail, an error message is shown
        //
```

```
        if(stsResult != TPCANStatus.PCAN_ERROR_OK)
            MessageBox.Show(GetFormatedError(stsResult));
    }

    private void btnParameterGet_Click(object sender, EventArgs e)
    {
        TPCANStatus stsResult;
        UInt32 iBuffer;

        // Gets a PCAN-Basic parameter value
        //
        switch (cbbParameter.SelectedIndex)
        {
            // The Device-Number of an USB channel will be retrieved
            //
            case 0:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_DEVICE_NUMBER, out iBuffer,
 sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The configured Device-Number is {0:X}h",iBuffer));
                break;
            // The activation status of the 5 Volt Power feature of a PC-card or USB will be retrieved
            //
            case 1:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_5VOLTS_POWER, out iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The 5-Volt Power of the USB/PC-Card is {0}", (iBuffer
== PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The activation status of the feature for automatic reset on BUS-OFF will be retrieved
            //
            case 2:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_BUSOFF_AUTORESET, out
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The automatic-reset on BUS-OFF is {0}", (iBuffer ==
PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The activation status of the CAN option "Listen Only" will be retrieved
            //
            case 3:
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_LISTEN_ONLY, out iBuffer,
sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The CAN-option Listen-Only is {0}", (iBuffer ==
PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The activation status for the feature for logging debug-information will be retrieved
            case 4:
                stsResult = PCANBasic.GetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_LOG_STATUS, out
iBuffer, sizeof(UInt32));
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                    IncludeTextMessage(string.Format("The feature for logging debug information is {0}", (
iBuffer == PCANBasic.PCAN_PARAMETER_ON) ? "ON" : "OFF"));
                break;
            // The current parameter is invalid
            //
            default:
                stsResult = TPCANStatus.PCAN_ERROR_UNKNOWN;
                MessageBox.Show("Wrong parameter code.");
                return;
        }

        // If the function fail, an error message is shown
        //
        if (stsResult != TPCANStatus.PCAN_ERROR_OK)
            MessageBox.Show(GetFormatedError(stsResult));
    }

    private void btnRead_Click(object sender, EventArgs e)
    {
        TPCANMsg CANMsg;
        TPCANTimestamp CANTimeStamp;
        TPCANStatus stsResult;
```

```
            // We execute the "Read" function of the PCANBasic
            //
            stsResult = PCANBasic.Read(m_PcanHandle, out CANMsg, out CANTimeStamp);
            if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                // We process the received message
                //
                ProcessMessage(CANMsg, CANTimeStamp);
            else
                // If an error occurred, an information message is included
                //
                IncludeTextMessage(GetFormatedError(stsResult));
        }

        private void btnGetVersions_Click(object sender, EventArgs e)
        {
            TPCANStatus stsResult;
            StringBuilder strTemp;
            string[] strArrayVersion;

            strTemp = new StringBuilder(256);

            // We get the vesion of the PCAN-Basic API
            //
            stsResult = PCANBasic.GetValue(PCANBasic.PCAN_NONEBUS, TPCANParameter.PCAN_API_VERSION, strTemp,
256);
            if (stsResult == TPCANStatus.PCAN_ERROR_OK)
            {
                IncludeTextMessage("API Version: " + strTemp.ToString());
                // We get the driver version of the channel being used
                //
                stsResult = PCANBasic.GetValue(m_PcanHandle, TPCANParameter.PCAN_CHANNEL_VERSION, strTemp, 256);
                if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                {
                    // Because this information contains line control characters (several lines)
                    // we split this also in several entries in the Information List-Box
                    //
                    strArrayVersion = strTemp.ToString().Split(new char[] { '\n' });
                    IncludeTextMessage("Channel/Driver Version: ");
                    for(int i =0; i < strArrayVersion.Length; i++)
                        IncludeTextMessage("     * " + strArrayVersion[i]);
                }
            }

            // If an error ccurred, a message is shown
            //
            if (stsResult != TPCANStatus.PCAN_ERROR_OK)
                MessageBox.Show(GetFormatedError(stsResult));
        }

        private void btnMsgClear_Click(object sender, EventArgs e)
        {
            // The information contained in the messages List-View
            // is cleared
            //
            lock (m_LastMsgsList.SyncRoot)
            {
                m_LastMsgsList.Clear();
                lstMessages.Items.Clear();
            }
        }

        private void btnInfoClear_Click(object sender, EventArgs e)
        {
            // The information contained in the Information List-Box
            // is cleared
            //
            lbxInfo.Items.Clear();
        }

        private void btnWrite_Click(object sender, EventArgs e)
        {
            TPCANMsg CANMsg;
            TextBox txtbCurrentTextBox;
            TPCANStatus stsResult;

            // We create a TPCANMsg message structure
```

```
            //
            CANMsg = new TPCANMsg();
            CANMsg.DATA = new byte[8];

            // We configurate the Message.  The ID (max 0x1FF),
            // Length of the Data, Message Type (Standard in
            // this example) and die data
            //
            CANMsg.ID = Convert.ToUInt32(txtID.Text, 16);
            CANMsg.LEN = Convert.ToByte(nudLength.Value);
            CANMsg.MSGTYPE = (chbExtended.Checked) ? TPCANMessageType.PCAN_MESSAGE_EXTENDED : TPCANMessageType.
PCAN_MESSAGE_STANDARD;
            // If a remote frame will be sent, the data bytes are not important.
            //
            if (chbRemote.Checked)
                CANMsg.MSGTYPE |= TPCANMessageType.PCAN_MESSAGE_RTR;
            else
            {
                // We get so much data as the Len of the message
                //
                txtbCurrentTextBox = txtData0;
                for (int i = 0; i < CANMsg.LEN; i++)
                {
                    CANMsg.DATA[i] = Convert.ToByte(txtbCurrentTextBox.Text, 16);
                    if (i < 7)
                        txtbCurrentTextBox = (TextBox)this.GetNextControl(txtbCurrentTextBox, true);
                }
            }

            // The message is sent to the configured hardware
            //
            stsResult = PCANBasic.Write(m_PcanHandle, ref CANMsg);

            // The message was successfully sent
            //
            if (stsResult == TPCANStatus.PCAN_ERROR_OK)
                IncludeTextMessage("Message was successfully SENT");
            // An error occurred.  We show the error.
            //
            else
                MessageBox.Show(GetFormatedError(stsResult));
        }

        private void btnReset_Click(object sender, EventArgs e)
        {
          TPCANStatus stsResult;

          // Resets the receive and transmit queues of a PCAN Channel.
          //
          stsResult = PCANBasic.Reset(m_PcanHandle);

          // If it fails, a error message is shown
          //
          if (stsResult != TPCANStatus.PCAN_ERROR_OK)
             MessageBox.Show(GetFormatedError(stsResult));
          else
             IncludeTextMessage("Receive and transmit queues successfully reset");
        }

        private void btnStatus_Click(object sender, EventArgs e)
        {
           TPCANStatus stsResult;
           String errorName;

           // Gets the current BUS status of a PCAN Channel.
           //
           stsResult = PCANBasic.GetStatus(m_PcanHandle);

           // Switch On Error Name
           //
           switch(stsResult)
           {
              case TPCANStatus.PCAN_ERROR_INITIALIZE:
                 errorName = "PCAN_ERROR_INITIALIZE";
                 break;
```

```
            case TPCANStatus.PCAN_ERROR_BUSLIGHT:
                errorName = "PCAN_ERROR_BUSLIGHT";
                break;

            case TPCANStatus.PCAN_ERROR_BUSHEAVY:
                errorName = "PCAN_ERROR_BUSHEAVY";
                break;

            case TPCANStatus.PCAN_ERROR_BUSOFF:
                errorName = "PCAN_ERROR_BUSOFF";
                break;

            case TPCANStatus.PCAN_ERROR_OK:
                errorName = "PCAN_ERROR_OK";
                break;

            default:
                errorName = "See Documentation";
                break;
        }

        // Display Message
        //
        IncludeTextMessage(String.Format("Status: {0} ({1:X}h)", errorName, stsResult));
    }
    #endregion

    #region Timer event-handler
    private void tmrRead_Tick(object sender, EventArgs e)
    {
        // Checks if in the receive-queue are currently messages for read
        //
        ReadMessages();
    }

    private void tmrDisplay_Tick(object sender, EventArgs e)
    {
        DisplayMessages();
    }
    #endregion

    #region Message List-View event-handler
    private void lstMessages_DoubleClick(object sender, EventArgs e)
    {
        // Clears the content of the Message List-View
        //
        btnMsgClear_Click(this, new EventArgs());
    }
    #endregion

    #region Information List-Box event-handler
    private void lbxInfo_DoubleClick(object sender, EventArgs e)
    {
        // Clears the content of the Information List-Box
        //
        btnInfoClear_Click(this, new EventArgs());
    }
    #endregion

    #region Textbox event handlers
    private void txtID_Leave(object sender, EventArgs e)
    {
        int iTextLength;
        uint uiMaxValue;

        // Calculates the text length and Maximum ID value according
        // with the Message Type
        //
        iTextLength = (chbExtended.Checked) ? 8 : 3;
        uiMaxValue = (chbExtended.Checked) ? (uint)0x1FFFFFF : (uint)0x7FF;

        // The Textbox for the ID is represented with 3 characters for
        // Standard and 8 characters for extended messages.
        // Therefore if the Length of the text is smaller than TextLength,
        // we add "0"
        //
```

```
        while (txtID.Text.Length != iTextLength)
            txtID.Text = ("0" + txtID.Text);

        // We check that the ID is not bigger than current maximum value
        //
        if (Convert.ToUInt32(txtID.Text, 16) > uiMaxValue)
            txtID.Text = string.Format("{0:X" + iTextLength.ToString() + "}", uiMaxValue);
    }

    private void txtID_KeyPress(object sender, KeyPressEventArgs e)
    {
        char chCheck;

        // We convert the Character to its Upper case equivalent
        //
        chCheck = char.ToUpper(e.KeyChar);

        // The Key is the Delete (Backspace) Key
        //
        if (chCheck == 8)
            return;
        // The Key is a number between 0-9
        //
        if ((chCheck > 47) && (chCheck < 58))
            return;
        // The Key is a character between A-F
        //
        if ((chCheck > 64) && (chCheck < 71))
            return;

        // Is neither a number nor a character between A(a) and F(f)
        //
        e.Handled = true;
    }

    private void txtData0_Leave(object sender, EventArgs e)
    {
        TextBox txtbCurrentTextbox;

        // all the Textbox Data fields are represented with 2 characters.
        // Therefore if the Length of the text is smaller than 2, we add
        // a "0"
        //
        if (sender.GetType().Name == "TextBox")
        {
            txtbCurrentTextbox = (TextBox)sender;
            while (txtbCurrentTextbox.Text.Length != 2)
                txtbCurrentTextbox.Text = ("0" + txtbCurrentTextbox.Text);
        }
    }
    #endregion

    #region Radio- and Check- Buttons event-handlers
    private void chbShowPeriod_CheckedChanged(object sender, EventArgs e)
    {
        // According with the check-value of this checkbox,
        // the recieved time of a messages will be interpreted as
        // period (time between the two last messages) or as time-stamp
        // (the elapsed time since windows was started)
        //
        lock (m_LastMsgsList.SyncRoot)
        {
            foreach (MessageStatus msg in m_LastMsgsList)
                msg.ShowingPeriod = chbShowPeriod.Checked;
        }
    }

    private void chbExtended_CheckedChanged(object sender, EventArgs e)
    {
        uint uiTemp;

        txtID.MaxLength = (chbExtended.Checked) ? 8 : 3;

        // the only way that the text length can be bigger als MaxLength
        // is when the change is from Extended to Standard message Type.
        // We have to handle this and set an ID not bigger than the Maximum
```

```
        // ID value for a Standard Message (0x7FF)
        //
        if (txtID.Text.Length > txtID.MaxLength)
        {
            uiTemp = Convert.ToUInt32(txtID.Text, 16);
            txtID.Text = (uiTemp < 0x7FF) ? string.Format("{0:X3}", uiTemp) : "7FF";
        }

        txtID_Leave(this, new EventArgs());
    }

    private void chbRemote_CheckedChanged(object sender, EventArgs e)
    {
        TextBox txtbCurrentTextBox;

        txtbCurrentTextBox = txtData0;

        // If the message is a RTR, no data is sent. The textbox for data
        // will be turned invisible
        //
        for (int i = 0; i < 8; i++)
        {
            txtbCurrentTextBox.Visible = !chbRemote.Checked;
            if (i < 7)
                txtbCurrentTextBox = (TextBox)this.GetNextControl(txtbCurrentTextBox, true);
        }
    }

    private void chbFilterExt_CheckedChanged(object sender, EventArgs e)
    {
        int iMaxValue;

        iMaxValue  = (chbFilterExt.Checked) ? 0x1FFFFFFF : 0x7FF;

        // We check that the maximum value for a selected filter
        // mode is used
        //
        if (nudIdTo.Value > iMaxValue)
            nudIdTo.Value = iMaxValue;

        nudIdTo.Maximum = iMaxValue;
        if (nudIdFrom.Value > iMaxValue)
            nudIdFrom.Value = iMaxValue;

        nudIdFrom.Maximum = iMaxValue;
    }

    private void rdbTimer_CheckedChanged(object sender, EventArgs e)
    {
        if (!btnRelease.Enabled)
            return;

        // According with the kind of reading, a timer, a thread or a button will be enabled
        //
        if (rdbTimer.Checked)
        {
            // Abort Read Thread if it exists
            //
            if (m_ReadThread != null)
            {
                m_ReadThread.Abort();
                m_ReadThread.Join();
                m_ReadThread = null;
            }

            // Enable Timer
            //
            tmrRead.Enabled = btnRelease.Enabled;
        }
        if (rdbEvent.Checked)
        {
            // Disable Timer
            //
            tmrRead.Enabled = false;
            // Create and start the tread to read CAN Message using SetRcvEvent()
            //
```

```
                System.Threading.ThreadStart threadDelegate = new System.Threading.ThreadStart(this.
    CANReadThreadFunc);
                m_ReadThread = new System.Threading.Thread(threadDelegate);
                m_ReadThread.IsBackground = true;
                m_ReadThread.Start();
            }
            if (rdbManual.Checked)
            {
                // Abort Read Thread if it exists
                //
                if (m_ReadThread != null)
                {
                    m_ReadThread.Abort();
                    m_ReadThread.Join();
                    m_ReadThread = null;
                }
                // Disable Timer
                //
                tmrRead.Enabled = false;
            }
            btnRead.Enabled = btnRelease.Enabled && rdbManual.Checked;
        }
        #endregion

        private void saveAsToolStripMenuItem_Click(object sender, EventArgs e)
        {
            Application.Restart();
        }

        #endregion

        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }

        #endregion

        private void helpToolStripMenuItem_Click(object sender, EventArgs e)
        {

        }
    }
}
```