Bus CAN et systèmes PSOC5

publié le 10/02/2013 - mis à jour le 17/10/2019

Descriptif :

Les composants PsoC5 intègrent un contrôleur CAN. Cet article décrit une mise en oeuvre permettant l'exploitation du bus. Le champ des exploitations est ensuite très vaste (développement de projet, modélisation de systèmes liés à l'automobile...)

Sommaire :

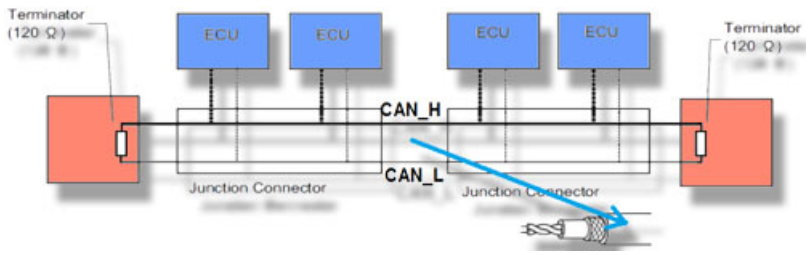
BUS CAN et PSOC5

1. Présentation

1.1 Généralités

Le Bus CAN (Control Area Network) est largement employé pour véhiculer des informations dans les systèmes embarqués. Sa robustesse en fait un bus de choix dans les domaines de l'automobile la marine et l'aviation.

Le bus CAN HS (High Speed) est un bus multi-maître qui permet la liaison entre un ensemble de nœuds (ECU : Electronic Control Unit). Le débit d'information peut atteindre 1Mbit/s.

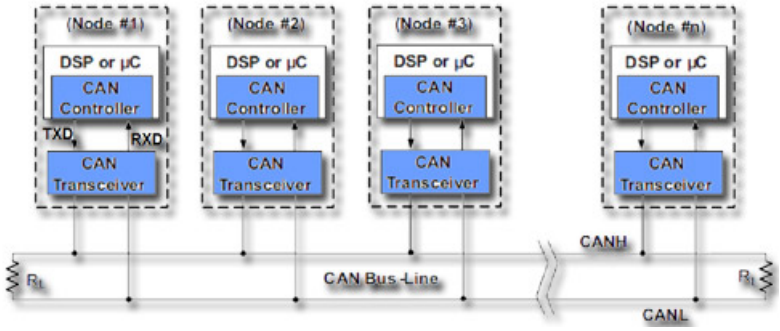


Structure du bus CAN

Les ECU sont réalisés à l'aide d'unités de traitements (microcontrôleurs, DSP ou ordinateurs) reliés à des ensembles "controller + transceiver". Le transceiver (coupleur différentiel) permet l'adaptation des niveaux électriques des tensions circulant sur le bus. Le contrôleur, quant à lui, a pour fonction de gérer les échanges de données sur le bus (priorité, erreur de données...).

Comme nous allons le voir dans la suite de l'article, le PSOC5 intègre la partie contrôleur.

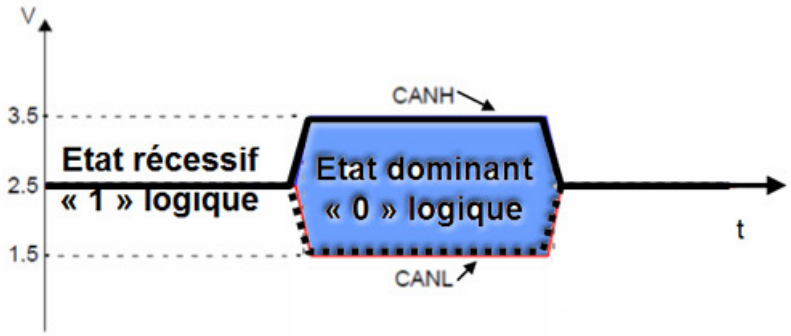
Il suffira juste de lui adjoindre un transceiver afin de faire communiquer ce microcontrôleur avec un autre environnement (l'exemple que je décris ci-après concerne la communication entre deux kits PSOC5).



1.2 Niveaux de tension, informations binaires et transceiver

La méthode de codage des bits composant la trame est de type « NRZ ». A chaque bit généré correspond un niveau de tension.

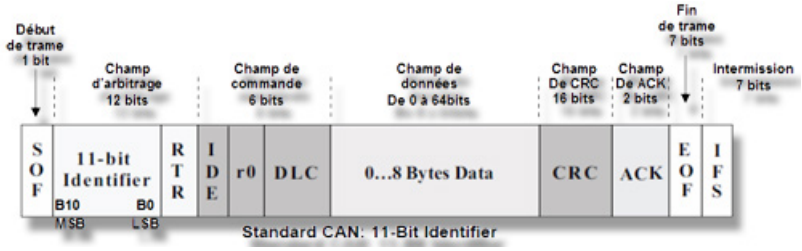
Les transceivers assurent la création des états Récessifs (1 logique) ou Dominants (0 logique) en appliquant sur le bus les niveaux de tension décrits ci-après. ***C'est la valeur de la tension différentielle appliquée sur le bus qui représente l'état souhaité*** .



1.3 Trame CAN

Le bus CAN est capable de véhiculer quatre types de trames :

- Trame de données (voir description ci-dessous)
- Trame de questionnement (remote frame)
- Trame d'erreur (error frame)
- Trame de surcharge (overload frame)

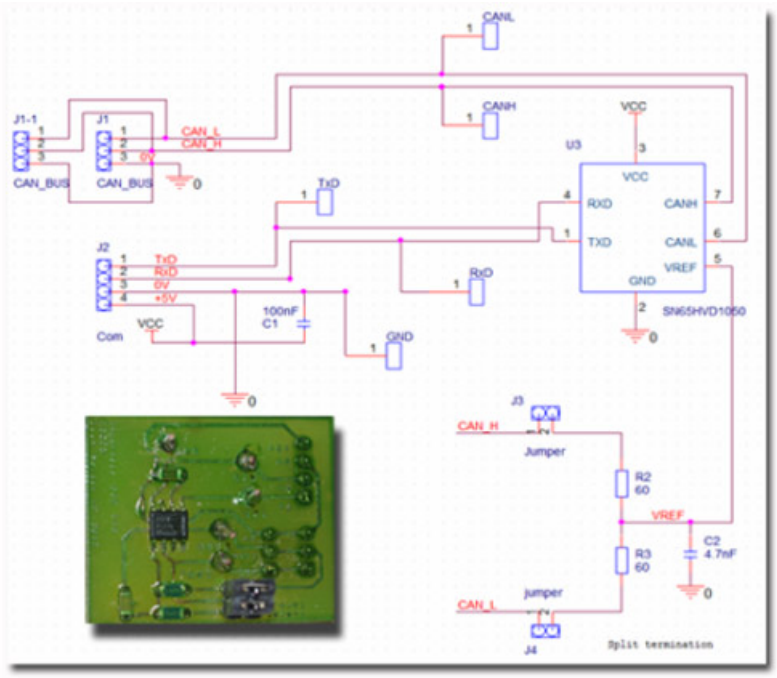


L'idée n'étant pas de commenter complètement le bus CAN j'arrête là la description du bus pour passer à son exploitation.

2. Exploitation du bus CAN avec le microcontrôleur PSOC5

2.1 Module transceiver

Afin d'interfacer correctement le contrôleur CAN présent dans le microcontrôleur, un module transceiver est nécessaire. Il est bien sur possible de le câbler directement sur la carte LAB du kit. J'ai préféré une réalisation offrant plus de modularité.



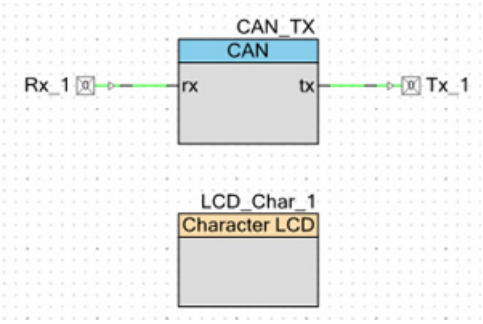
Le transceiver employé (TJA1050) a été implanté sur un circuit imprimé sur lequel existe la possibilité de placer les résistances de terminaison du bus (2 x 60 Ohms). Les documents nécessaires à la réalisation de cette interface sont accessibles :

Dossier de fabrication : schémas, typons, documentation technique. (Zip de 318.1 ko)

L'exploitation proposée consiste à réaliser un émetteur ainsi qu'un récepteur. Toutes les déclinaisons/exploitations sont ensuite possibles.

2.2 L'émetteur, construction du projet

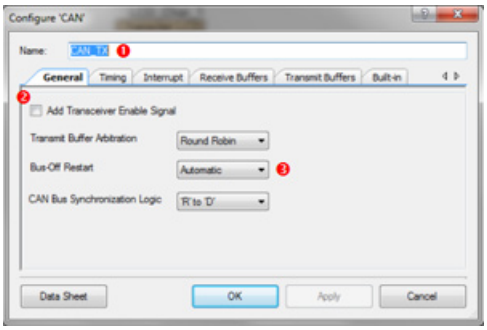
Le top design (PSOC Creator 1.0)

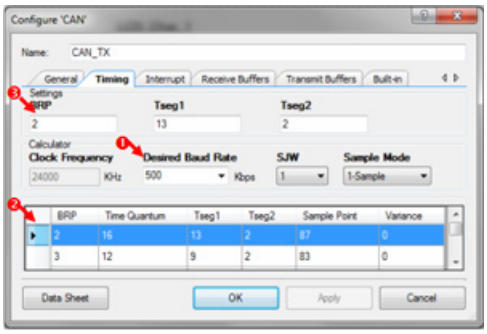


Il faut ajouter au top design le composant contrôleur CAN (renommé ici CAN_TX).

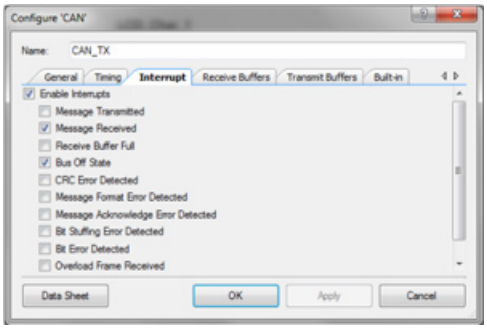
Il faut ensuite le configurer.
Arbitrairement le choix a été fixé à une vitesse de transmission de 500kb/s et 1 octet de champ de donnée.

1. Imposer le nom du contrôleur CAN
2. ôter l'option du signal de validation du transceiver (on ne se servira que des broches RX et TX))
3. Sélectionner l'option Automatic

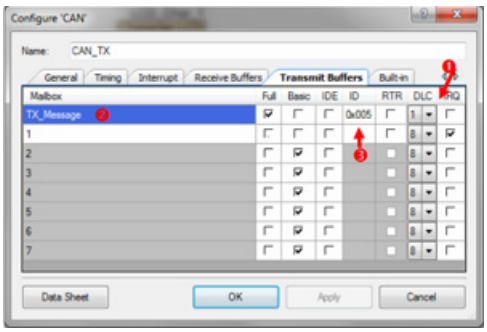




1. Choisir la vitesse de transmission
2. Sélectionner le timing désiré (double clic). Les champs de la ligne "3" seront complétés automatiquement.



Pas de manipulation ici, sauf si des besoins d'interruptions supplémentaires sont nécessaires.



1. Modification de DLC = 1 (longueur de donnée à transmettre ici, 1 octet)
2. Changement du nom de la case "mailbox" (champ de données) à transmettre.
3. Identification du noeud emetteur (codé sur 11bits _Full_ en mode IDE il est possible de coder l'identifiant sur 29 bits), choisi arbitrairement à 0x005.

Chaque "MailBox" peut contenir jusqu'à 8 octets à transmettre. Les variables permettant de compléter ces "mailboxes" sont identifiées ainsi dans la fonction CAN_TX_TX_RX_func.c (voir après) :

Pour la "mailbox0" (ici renommée TX_Message), la variable "container" est : CAN_TX_TX_DATA_BYTE1(0).

Elle contient le premier octet à transmettre. Cela nous suffit puisque DLC=1. Si on avait désiré transmettre plus d'octets

(DLC = 2 par exemple), il aurait fallu employer deux variables CAN_TX_TX_DATA_BYTE1(0) et CAN_TX_TX_DATA_BYTE2(0).

	Rx_1	P0[0]
	Tx_1	P0[1]
	\LCD_Char_1:LCDPort\[6:0]	P2[6:0]

A ce stade il convient d'assigner les broches de l'afficheur ainsi que du contrôleur.

Il faut à présent construire le projet "Shift F6" et compléter le programme principal ainsi que la fonction CAN_TX_TX_RX_func.c qui a été générée

2.3 Programmation de l'émetteur

Le projet complet du transmetteur.

Fichier projet complet de la partie émetteur pour bus CAN (sous PSOC creator1) (Zip de 2.9 Mo)

2.3.1 main.c

```
 *BUS CAN TEST Transmission
 *Laurent Proust
 * ========================================
*/
#include <device.h>
//initialisation de la variable contenant les données
// à transmettre
uint8 CAN_TX_Data=0;

/*****************************
*        ATTENTION           *
*****************************

Dans la fonction CAN_TX_RX_RX_func.c, il est nécessaire d'ajouter les lignes :

`#START TX_RX_FUNCTION`
extern uint8 CAN_TX_Data;
`#END`

dans l'entête et

`#START MESSAGE_TX_Message_TRASMITTED`
CAN_TX_TX_DATA_BYTE1(0)=CAN_TX_Data;
`#END`

 dans la fonction uint8 CAN_TX_SendMsgTX_Message(void)

void main()
{
    //Déclaration d'un tableau de données à envoyer via bus can (ici du texte)
    char texte[]="TEST SIN";
    char i=0;//déclaration variable de boucle

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    CAN_TX_Start();//lancement controleur CAN
    LCD_Char_1_Start();//lancement afficheur
    CYGlobalIntEnable;/* Uncomment this line to enable global interrupts. */

    LCD_Char_1_PrintString("TEST BUSCAN TX");
    CyDelay(2000);//delai de 2s
    for(;;)
    {
    LCD_Char_1_ClearDisplay();//effacement écran
    LCD_Char_1_PrintString("DATA TX:");
    LCD_Char_1_Position(0,8);

        for (i=0;i<=7;i++)//boucle pour les huit caractères du tableau
        {
            CAN_TX_Data=texte[i];//placer les caractères du texte dans la variable de "transmission"
            CAN_TX_SendMsgTX_Message();//envoie de la donnée mailbox0 TX_Message (DLC = 1 => 1 octet)
            LCD_Char_1_PutChar(CAN_TX_Data);//affichage de la donnée transmise
            CyDelay(100);//attente avant d'envoyer un nouveau caractère
        }
    }
}
```

2.3.2 Modification de la fonction CAN_TX_TX_RX_func.c

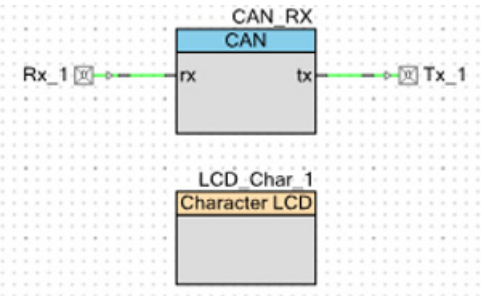
```
/*******************************************************************************
* File Name: CAN_TX_TX_RX_func.c
* Version 1.50
*
* Description:
*  There are fucntions process "Full" Receive and Transmit mailboxes:
*     - CAN_TX_SendMsg0-7();
*     - CAN_TX_ReceiveMsg0-15();
*  Transmition of message, and receive routine for "Basic" mailboxes:
*     - CAN_TX_SendMsg();
*     - CAN_TX_TxCancel();
*     - CAN_TX_ReceiveMsg();
*
*  Note:
*   None
*
********************************************************************************
* Copyright 2008-2010, Cypress Semiconductor Corporation.  All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*******************************************************************************/

#include "CAN_TX.h"

/* `#START TX_RX_FUNCTION` */
extern uint8 CAN_TX_Data;
/* `#END` */
```

```
uint8 CAN_TX_SendMsgTX_Message(void)
{
    uint8 result = CYRET_SUCCESS;

    if ((CAN_TX_TX[0u].txcmd.byte[0u] & CAN_TX_TX_REQUEST_PENDING) ==
        CAN_TX_TX_REQUEST_PENDING)
    {
        result = CAN_TX_FAIL;
    }
    else
    {
        /* `#START MESSAGE_TX_Message_TRASMITTED` */
        CAN_TX_TX_DATA_BYTE1(0)=CAN_TX_Data;
        /* `#END` */

        CY_SET_REG32((reg32 *) &CAN_TX_TX[0u].txcmd, CAN_TX_SEND_MESSAGE);
    }

    return(result);
}
```

A ce stade il reste à programmer le composant. C'est terminé pour la partie émetteur.

2.4 Le récepteur, construction du projet

Le top design (PSOC Creator 1.0)



Il faut ajouter au top design le composant contrôleur CAN (renommé ici CAN_RX).

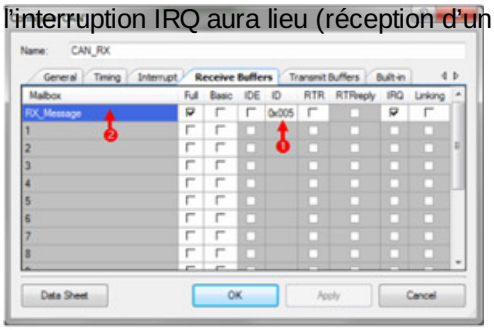
Il faut ensuite le configurer afin qu'il puisse communiquer avec le module émetteur déjà mis au point.
(vitesse de transmission de 500kb/s et 1 octet de champ de donnée)

Les trois premières parties de la configuration du contrôleur sont identiques à celles de l'émetteur (General, Timing, Interrupt). ***Seule la partie "Receive Buffer" est à modifier*** .

1. L'identifiant choisi est celui de l'émetteur dont on souhaite recevoir les données...donc 0x005.
2. Le nom "0" du champ "mailbox" est changé pour RX_Message. Cela permettra de créer une fonction de traitement de réception facilement identifiable.

Cette fonction sera nommée (on le verra après) "CAN_RX_ReceivedMsgRX_Message()" et sera exécuté dès que

l'interruption IRQ aura lieu (réception d'un octet).



C'est dans la variable CAN_RX_RX_DATA_BYTE1(0), correspondante à la "mailbox0" que se retrouvera la donnée reçue.

	Rx_1	P0[0]
	Tx_1	P0[1]
	\LCD_Char_1:LCDPort\[6:0]	P2[6:0]

A ce stade il convient d'assigner les broches de l'afficheur ainsi que du contrôleur.

Il faut à présent construire le projet "Shift F6" et compléter le programme principal ainsi que la fonction CAN_RX_TX_RX_func.c qui a été générée.

2.5 Programmation du récepteur

Le projet complet du récepteur.

Fichier projet complet de la partie réceptrice bus CAN (sous PSOC creator1) (Zip de 1.6 Mo)

2.5.1 main.c

```
/* ========================================
 *BUS CAN TEST RECEPTION
 *Laurent Proust
 * ========================================
*/
#include <device.h>
//initialisation de la variable contenant les données
// à recevoir
uint8 CAN_RX_Data=0;

//initialisation de la variable permettant de savoir si une données a été reçue
uint8 Reception_OK_CAN=0;

/*****************************
*         ATTENTION          *
*****************************

Dans la fonction CAN_RX_TX_RX_func.c, il est nécessaire d'ajouter les lignes :

`#START TX_RX_FUNCTION`
extern uint8 CAN_TX_Data;
extern uint8 Reception_OK_CAN;
`#END`
dans l'entête.

La fonction void CAN_RX_ReceiveMsgRX_Message(void)
doit contenir

`#START MESSAGE_0_TRASMITTED`
Reception_OK_CAN=1;
CAN_RX_Data=CAN_RX_RX_DATA_BYTE1(0);
`#END`
```

```c
void main()
{
    char i=0;//déclaration variable de boucle

    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

    CAN_RX_Start();//lancement controleur CAN
    LCD_Char_1_Start();//lancement afficheur
    CYGlobalIntEnable;/* Uncomment this line to enable global interrupts. */

    LCD_Char_1_PrintString("TEST BUSCAN RX");
    CyDelay(2000);//delai de 2s

    LCD_Char_1_ClearDisplay();//effacement écran
    LCD_Char_1_PrintString("DATA RX:");
    LCD_Char_1_Position(0,8);

    for(;;)
    {

        if (Reception_OK_CAN==1)//si donnée reçue
        {
            i++;//détection d'une réception
            LCD_Char_1_PutChar(CAN_RX_Data);//affichage de la donnée reçue
            Reception_OK_CAN=0;//annulation et préparation réception d'une nouvelle donnée
                if (i>=8)//initialisation position afficheur
                (LCD_Char_1_Position(0,8);LCD_Char_1_PrintString("       ");//effacement
                LCD_Char_1_Position(0,8);LCD_Char_1_PutChar(CAN_RX_Data);//affichage du dernier caractere reçu
                i=0;//réinitialisation de i
                )
        }
    }
}

/* [] END OF FILE */
```

2.5.2 Modification de la fonction CAN_RX_TX_RX_func.c

```c
/*******************************************************************************
* File Name: CAN_RX_TX_RX_func.c
* Version 1.50
*
* Description:
*  There are fucntions process "Full" Receive and Transmit mailboxes:
*     - CAN_RX_SendMsg0-7();
*     - CAN_RX_ReceiveMsg0-15();
*  Transmition of message, and receive routine for "Basic" mailboxes:
*     - CAN_RX_SendMsg();
*     - CAN_RX_TxCancel();
*     - CAN_RX_ReceiveMsg();
*
*  Note:
*   None
*
********************************************************************************
* Copyright 2008-2010, Cypress Semiconductor Corporation.  All rights reserved.
* You may use this file only in accordance with the license, terms, conditions,
* disclaimers, and limitations in the end user license agreement accompanying
* the software package with which this file was provided.
*******************************************************************************/

#include "CAN_RX.h"

/* `#START TX_RX_FUNCTION` */
extern uint8 CAN_RX_Data;
extern uint8 Reception_OK_CAN;
/* `#END` */

    ------------------------------------------

    void CAN_RX_ReceiveMsgRX_Message(void)
    {
        /* `#START MESSAGE_RX_Message_RECEIVED` */
        Reception_OK_CAN=1;
        CAN_RX_Data=CAN_RX_RX_DATA_BYTE1(0);
        /* `#END` */

            CAN_RX_RX[0u].rxcmd.byte[0u] |= CAN_RX_RX_ACK_MSG;

    }
```

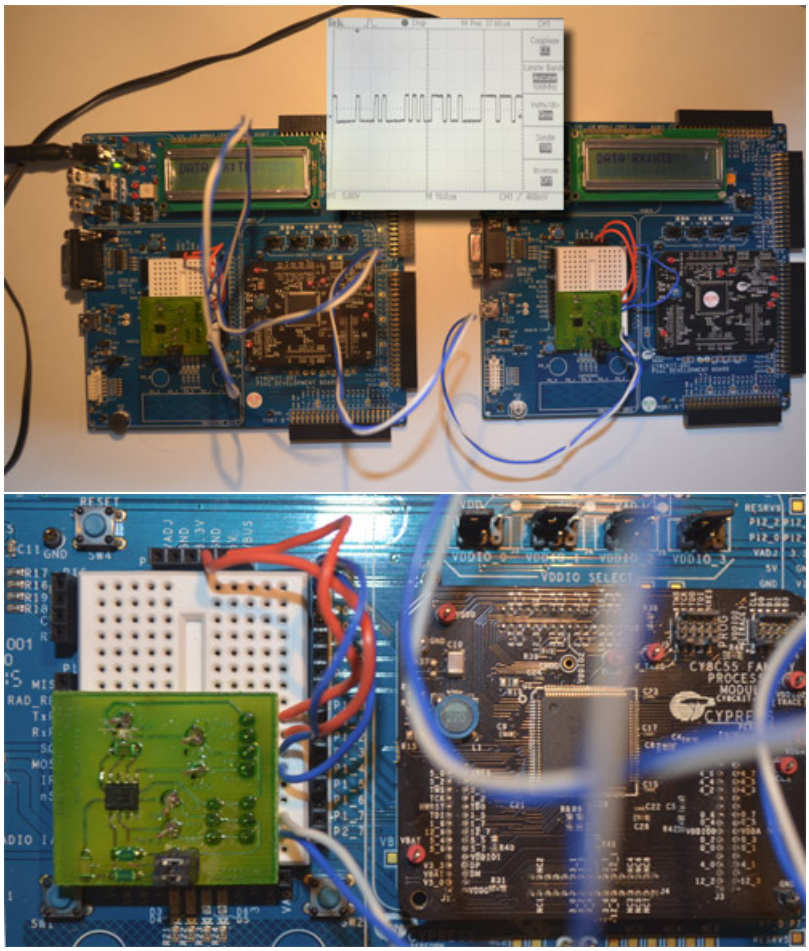
A ce stade il reste à programmer le composant. C'est terminé pour la partie récepteur.

3. Au Final

Connecter les transceivers sur les deux kits ainsi qu'un câble CAN (ce qui n'est pas le cas sur la photo...). Tout devrait fonctionner parfaitement.

Nota : Comme les deux noeuds (émetteur et récepteur) terminent le bus, j'ai activé les résistances de terminaison (120 Ohms) des modules transceivers.

Quelques photographies...



Laurent Proust 2013