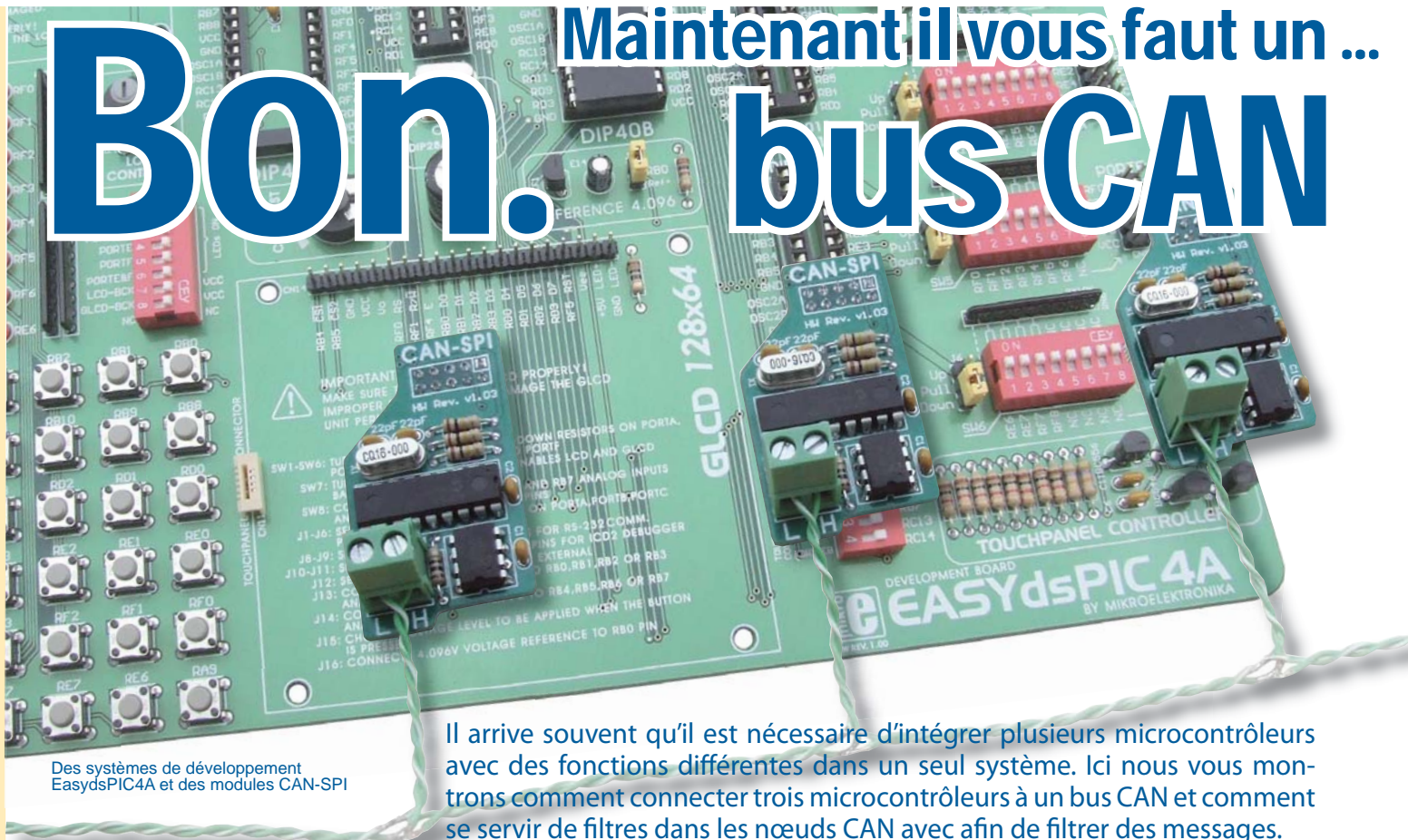


Bon. Maintenant il vous faut un ... bus CAN



Des systèmes de développement EasydsPIC4A et des modules CAN-SPI

par Zoran Ristic
MikroElektronika - Software Department

Lorsque plusieurs périphériques se partagent le même bus, il convient de définir la façon comment accéder à ce bus. Le protocole CAN décrit avec précision et en détail la connexion de plusieurs dispositifs à un bus, c'est un bus très répandu dans l'industrie. Le protocole définit principalement la présence d'accès au bus et résout le problème de collision au niveau matériel dans le cas où plusieurs périphériques commenceraient à communiquer en même temps.

Matériel

Cet exemple montre un bus CAN configuré de sorte que le premier dispositif envoie des messages à ID 0x10 et 0x11, tandis que le deuxième et le troisième dispositif envoient des messages respectivement à ID 0x12 et 0x13. Nous allons aussi configurer les nœuds CAN de façon à ce que le deuxième nœud ne réponde que à des messages entrants à ID de 0x10, tandis que le troisième répond seulement à ceux à ID 0x11. En conséquence, le premier dispositif est configuré pour recevoir des messages à ID 0x12 et 0x13 (Figure 2). Le filtrage de messages est facile à implémenter grâce à la fonction `CANSPISetFilter` qui configure les registres du microcontrô-

leur et du module CAN SPI. Il arrive souvent qu'il est nécessaire d'intégrer plusieurs microcontrôleurs avec des fonctions différentes dans un seul système. Ici nous vous montrons comment connecter trois microcontrôleurs à un bus CAN et comment se servir de filtres dans les nœuds CAN avec afin de filtrer des messages.

leur et du module CAN SPI.

En général, le protocole CAN n'a pas besoin d'un maître. Toutefois, pour faciliter la compréhension de cet exemple tout en lui conservant son objectif général, nous autorisons seulement le premier périphérique à initier la communication, les deux autres périphériques ne font que répondre.

Logiciel

Si un message est envoyé, le nœud maître laisse un temps de réponse suffisant au nœud appelé. Dans le cas où un nœud à distance ne répondrait pas dans le temps prévu, le maître signale une erreur dans le message actuel et continue à appeler les autres nœuds (Figure 3). Dans le cas où un nœud répondrait en même temps qu'un autre, il y aurait une collision sur le bus. Le protocole CAN prescrit dans ce cas que le nœud

émettant le message ayant la plus basse priorité se retire du bus, ce qui permet au nœud émettant un message à priorité plus élevée de continuer sa transmission.

Comme mentionné ci-dessus, nous utiliserons un module interne SPI du microcontrôleur pour transférer les données au bus CAN. L'utilisation du module interne SPI du microcontrôleur offre certains avantages : la possibilité de générer une interruption pendant l'envoi et la réception de données ; le module SPI opère indépendamment des autres périphériques et est facile à mettre en œuvre. La bibliothèque CAN SPI vous permet de paramétrer le mode opératoire du bus CAN et des filtres du nœud, de lire les données depuis la mémoire tampon du module CAN SPI, etc.

Cet exemple utilise aussi les LED connectées au microcontrôleur et qui

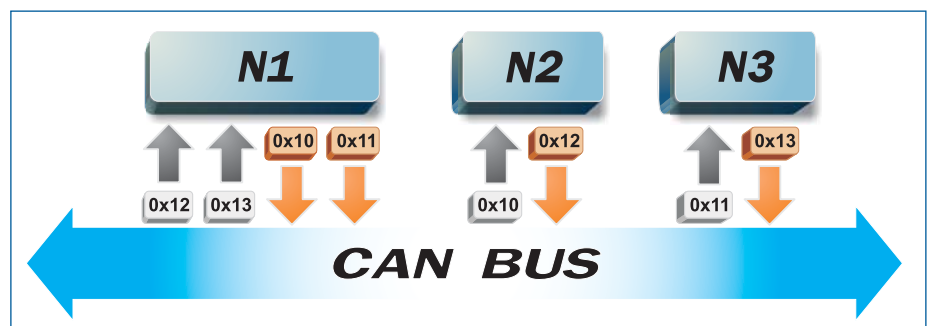


Figure 1. Filtrage de messages.

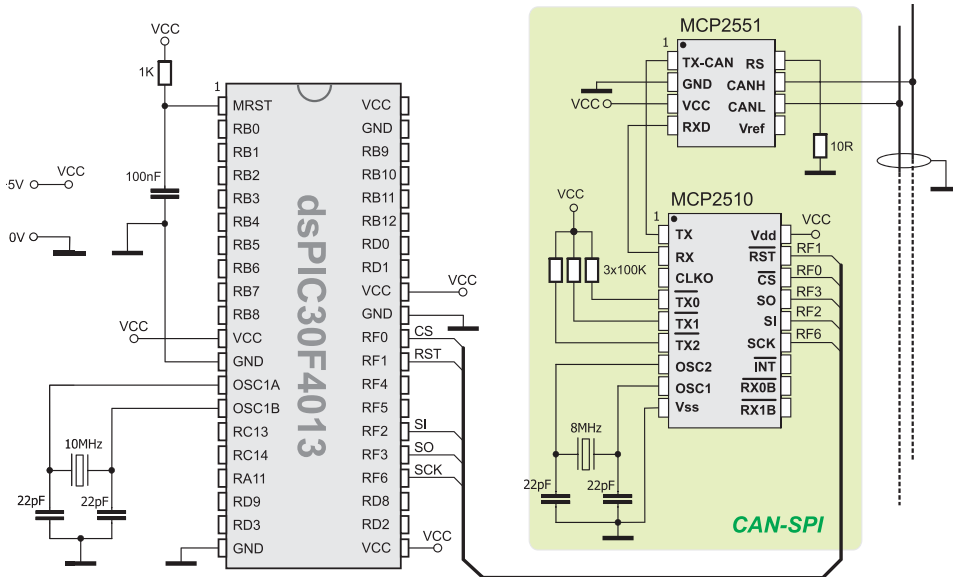


Schéma 1. Connexion du module CAN-SPI au dsPIC30F4013.

indiquent si le bus fonctionne correctement. Si le nœud 2 répond à l'appel du nœud 1, les LED du PORTB seront automatiquement allumées. Si le nœud 3 répond à l'appel, les LED du PORTD seront allumées. Le code source pour les trois nœuds est inclus dans l'exemple. Afin de créer un fichier HEX personnalisé pour chaque nœud, il suffit d'écrire une unique directive DEFINE dans en tête de l'exemple.

Programme montrant le fonctionnement d'un bus CAN.

```

char Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags;
// can flags
char Rx_Data_Len;
// received data length in bytes
char RxTx_Data[8];
// can rx/tx data buffer
char Msg_Rcvd;
// reception flag
long Tx_ID, Rx_ID;
// can rx and tx ID
char ErrorCount;

void main(){
    ADPCFG = 0xFFFF;
    // Configure analog pins as digital I/O
    PORTB = 0; TRISB = 0;
    // Initialize ports
    PORTD = 0; TRISD = 0;
    TRISF = 0;
    ErrorCount = 0;
    Can_Init_Flags = 0; Can_Send_Flags = 0; Can_Rcv_Flags = 0;

    Can_Send_Flags = CANSPI_TX_PRIORITY_0 &
    // form value to be used
    CANSPI_TX_XTD_FRAME &
    // with CANSPIWrite
    CANSPI_TX_NO_RTR_FRAME;

    Can_Init_Flags = CANSPI_CONFIG_SAMPLE_THRICE &
    // form value to be used CANSPI_
    CONFIG_SAMPLE_THRICE
    CANSPI_CONFIG_PHSSEG2_PRG_ON &
    // with CANSPIInit
    CANSPI_CONFIG_XTD_MSG &
    CANSPI_CONFIG_DBL_BUFFER_ON &
    CANSPI_CONFIG_VALID_XTD_MSG;

    SPI1_Init();
    CANSPIInit(1,3,3,3,1,Can_Init_Flags, &PORTF, 1, &PORTF,
0);
    // initialize external CANSPI module
    // initialize SPI module
    CANSPISetOperationMode(CANSPI_MODE_CONFIG, 0xFF);
    // set CONFIGURATION mode
    CANSPISetMask(CANSPI_MASK_B1, -1, CANSPI_CONFIG_XTD_
MSG);
    // set all mask1 bits to ones
    CANSPISetMask(CANSPI_MASK_B2, -1, CANSPI_CONFIG_XTD_
MSG);
    // set all mask2 bits to ones

    CANSPISetFilter(CANSPI_FILTER_B2_F4, 0x12, CANSPI_
CONFIG_XTD_MSG);
    // Node1 accepts messages with ID 0x12
    CANSPISetFilter(CANSPI_FILTER_B1_F1, 0x13, CANSPI_
CONFIG_XTD_MSG);
    // Node1 accepts messages with ID 0x13

    CANSPISetOperationMode(CANSPI_MODE_NORMAL, 0xFF);
    // set NORMAL mode
    RxTx_Data[0] = 0x40;
    // set initial data to be sent

    Tx_ID = 0x10;
    // set transmit ID for CAN message

    CANSPIWrite(Tx_ID, &RxTx_Data, 1, Can_Send_Flags);
    // Node1 sends initial message

    while (1)
        // endless loop
    {
        Msg_Rcvd = CANSPIRead(&Rx_ID, &RxTx_Data, &Rx_Data_
Len, &Can_Rcv_Flags);
        // attempt receive message
        if (Msg_Rcvd) {
            // if message is received then check id

            if (Rx_ID == 0x12)
                // check ID
                PORTB = RxTx_Data[0];
                // output data at PORTB
            else
                PORTD = RxTx_Data[0];
                // output data at PORTD
            delay_ms(50);
            // wait for a while between messages
            CANSPIWrite(Tx_ID, &RxTx_Data, 1, Can_Send_Flags
);
            // send one byte of data
            Tx_ID++;
            // switch to next message
            if (Tx_ID > 0x11) Tx_ID = 0x10;
            // check overflow
        }
        else {
            // an error occurred, wait for a while

            ErrorCount++;
            // increment error indicator
            Delay_ms(10);
            // wait for 10ms
            if (ErrorCount > 10) {
                // timeout expired - process errors
                ErrorCount = 0;
                // reset error counter
                Tx_ID++;
                // switch to another message
                if (Tx_ID > 0x11) Tx_ID = 0x10;
                // check overflow
                CANSPIWrite(Tx_ID, &RxTx_Data, 1, Can_Send_
Flags);
                // send new message
            }
        }
    }
}
    
```

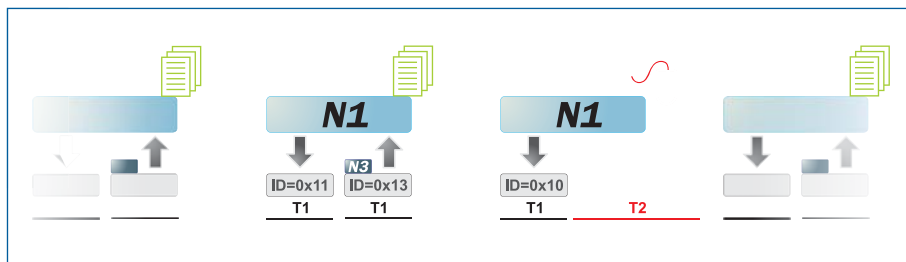


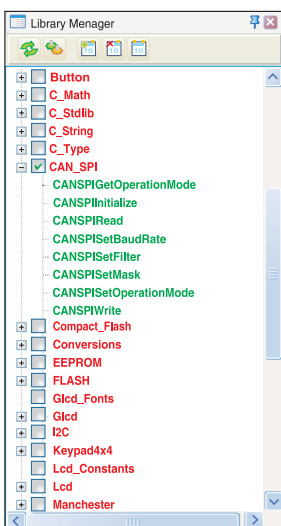
Figure 2. Exemple de communication.

Nous avons décrit ici une façon de connecter des microcontrôleurs au bus CAN. Nous avons aussi décrit comment détecter des erreurs à l'aide du protocole de communication dans le cas où un nœud à distance ne répondrait pas comme souhaité, comment filtrer des messages, ainsi que la procédure de communication utilisée en général sur le bus CAN.

mikroC for dsPIC® éditeur de librairie avec des librairies prêtes à l'emploi telles que: CAN_SPI, GLCD, Ethernet, etc.

Fonctions utilisées dans ce programme

- CANSPIGetOperationMode() lire le mode de fonctionnement actuel
- CANSPIInitialize()* Initialiser le module CANSPI
- CANIRead()* Lire un message
- CANSPISetBaudRate() Paramétrer la vitesse CANSPI
- CANSPISetFilter()* Configurer un filtre de messages
- CANSPISetMask()* Configurer le filtrage avancé
- CANSPISetOperationMode()* Choisir mode de fonctionnement
- CANSPIWrite()* Ecrire un message
- * fonctions de la librairie CANSPI utilisées dans le programme
- Autres fonctions de mikroC for dsPIC utilisées dans le programme:
 - Delay_us()
 - SPI1_init()
 - SPI1_read()



GO TO Le programme de cet exemple en C, BASIC et PASCAL pour microcontrôleurs dsPIC®, ainsi que tous les programmes écrits pour les microcontrôleurs PIC®, 8051® et AVR® sont disponibles sur notre site Internet : www.mikroe.com/en/article/

