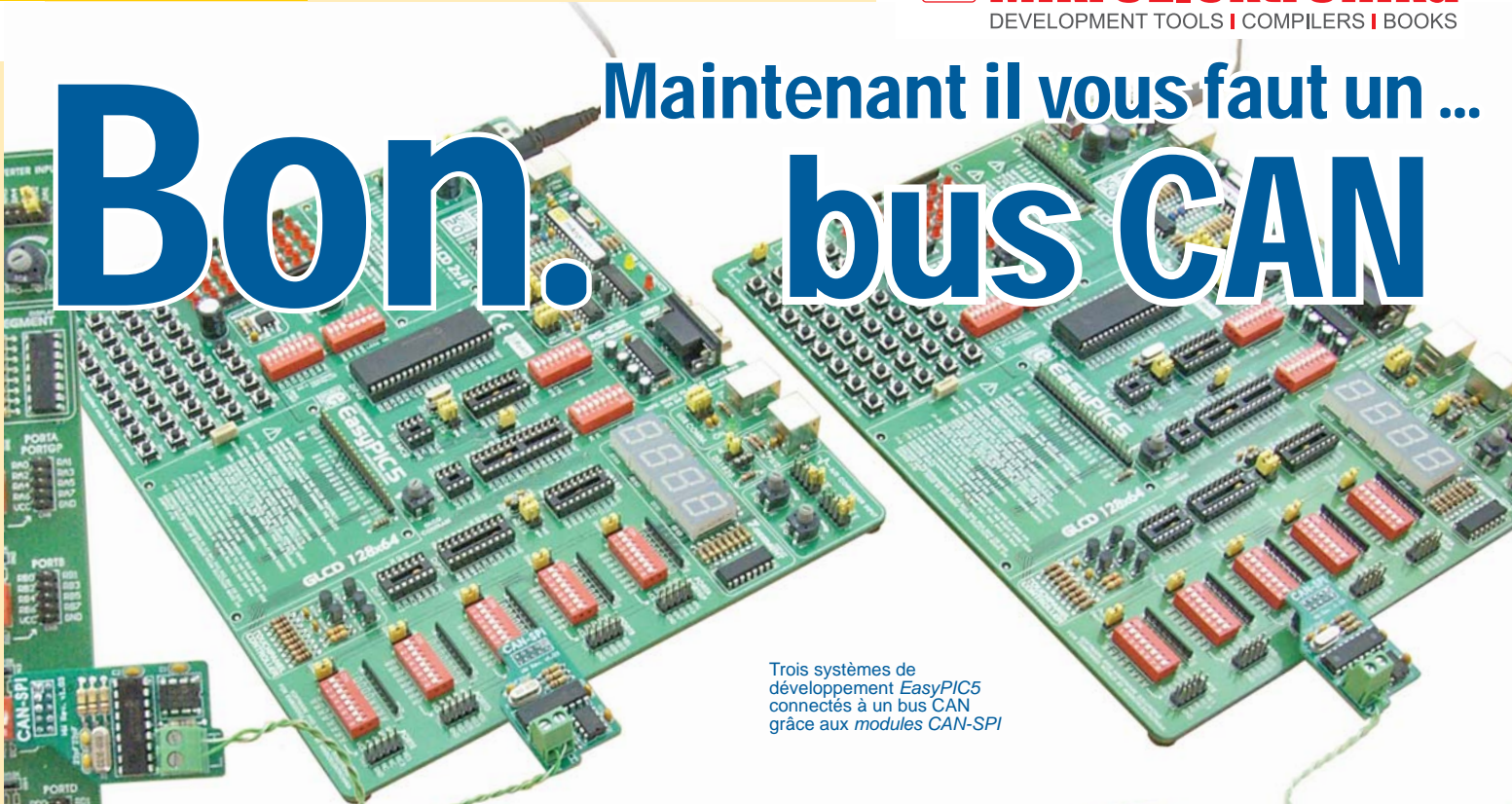


Bon. Maintenant il vous faut un ... bus CAN



Trois systèmes de développement EasyPIC5 connectés à un bus CAN grâce aux modules 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.

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.

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

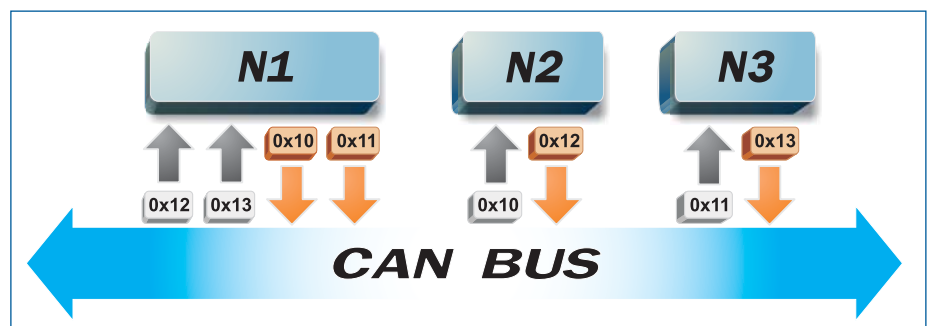


Figure 1. Filtrage de messages.

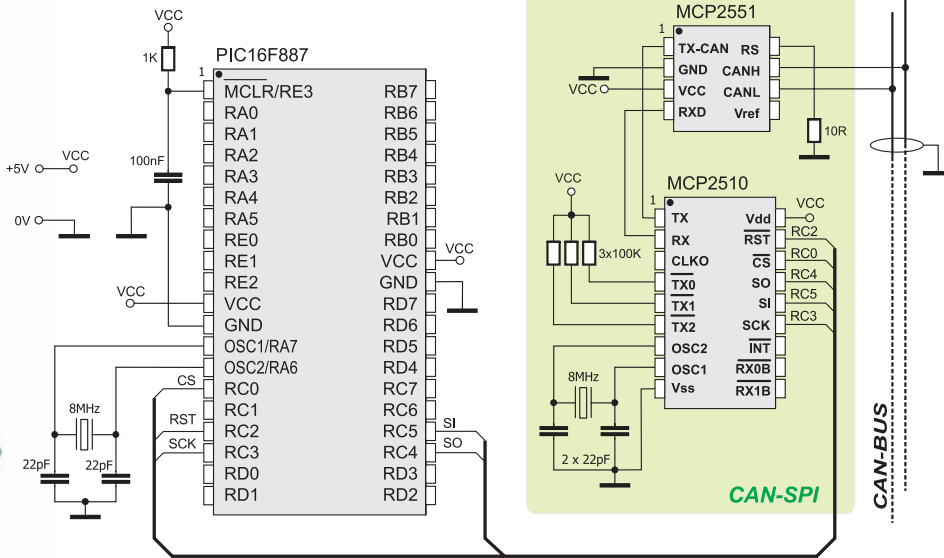


Schéma 1. Connexion du module CAN-SPI au PIC16F887.

Programme montrant le fonctionnement d'un bus CAN.

```
#define NODE1 // Uncomment this line to build HEX for Node 1
// #define NODE2 // Uncomment this line to build HEX for Node 2
// #define NODE3 // Uncomment this line to build HEX for Node 3
char Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags; // Can flags
char RxDx_Data[8]; // Received data length in bytes
char RxDx_Data[8]; // Can rx/tx data buffer
char Msg_Rcvd; // Reception flag
long Tx_ID, Rx_ID; // Can rx and tx ID
char ErrorCount;

// CANSPI module connections
sbit CanSpi_CS at RC0_bit; // Chip select (CS) pin for CANSPI board
sbit CanSpi_CS_Direction at TRISC0_bit; // Direction register for CS pin
sbit CanSpi_Rst at RC2_bit; // Reset pin for CANSPI board
sbit CanSpi_Rst_Direction at TRISC2_bit; // Direction register for Reset pin

// End CANSPI module connections

void main() {
  ANSEL = 0; ANSELH = 0; // Configure analog pins as digital I/O
  PORTB = 0; TRISB = 0; // Initialize ports
  PORTD = 0; TRISD = 0;
  ErrorCount = 0; // Error flag
  Can_Init_Flags = 0; Can_Send_Flags = 0; Can_Rcv_Flags = 0; // Clear flags

  Can_Send_Flags = _CANSPI_TX_PRIORITY_0 & // Form value to be used
    _CANSPI_TX_XID_FRAME & // with CANSPIwrite
    _CANSPI_TX_NO_RTR_FRAME;

  Can_Init_Flags = _CANSPI_CONFIG_SAMPLE_THRICE & // Form value to be used
    _CANSPI_CONFIG_PHSIG2_PRG_ON & // with CANSPIInit
    _CANSPI_CONFIG_XID_MSG &
    _CANSPI_CONFIG_DBL_BUFFER_ON &
    _CANSPI_CONFIG_VALID_XID_MSG;

  SPI1_Init(); // Initialize SPI module
  CANSPIInitialize(1, 3, 3, 1, Can_Init_Flags);
  // Initialize external CANSPI module
  CANSPISetOperationMode(_CANSPI_MODE_CONFIG, 0xFF);
  // Set CONFIGURATION mode
  CANSPISetMask(_CANSPI_MASK_B1, -1, _CANSPI_CONFIG_XID_MSG);
  // Set all mask1 bits to ones
  CANSPISetMask(_CANSPI_MASK_B2, -1, _CANSPI_CONFIG_XID_MSG);
  // Set all mask2 bits to ones
  #ifdef NODE1
  CANSPISetFilter(_CANSPI_FILTER_B2_F4, 0x12, _CANSPI_CONFIG_XID_MSG);
  // Node1 accepts messages with ID 0x12
  CANSPISetFilter(_CANSPI_FILTER_B1_F1, 0x13, _CANSPI_CONFIG_XID_MSG);
  // Node1 accepts messages with ID 0x13
  #else
  CANSPISetFilter(_CANSPI_FILTER_B2_F2, 0x10, _CANSPI_CONFIG_XID_MSG);
  // Node2 and Node3 accept messages with ID 0x10
  CANSPISetFilter(_CANSPI_FILTER_B1_F2, 0x11, _CANSPI_CONFIG_XID_MSG);
  // Node2 and Node3 accept messages with ID 0x11
  #endif
  CANSPISetOperationMode(_CANSPI_MODE_NORMAL, 0xFF); // Set NORMAL mode
  RxDx_Data[0] = 0x40; // Set initial data to be sent
  #ifdef NODE1
  Tx_ID = 0x10; // Set transmit ID for CAN message
  #endif
  #ifdef NODE2
  Tx_ID = 0x12; // set transmit ID for CAN message
  #endif
  #ifdef NODE3
  Tx_ID = 0x13; // Set transmit ID for CAN message
  #endif
  #ifdef NODE1
  CANSPIWrite(Tx_ID, &RxDx_Data, 1, Can_Send_Flags);
  // Node1 sends initial message
  #endif
  while (1) // Endless loop
  {
    Msg_Rcvd = CANSPIRead(&Rx_ID, &RxDx_Data, &RxDx_Data_Len, &Can_Rcv_Flags);
    // Attempt receive message
    if (Msg_Rcvd) { // If message is received then check id
      #ifdef NODE1
      if (Rx_ID == 0x12) // Check ID
        PORTB = RxDx_Data[0]; // Output data at PORTB
      else
        PORTD = RxDx_Data[0]; // Output data at PORTD
      delay_ms(50); // Wait for a while between messages
      CANSPIWrite(Tx_ID, &RxDx_Data, 1, Can_Send_Flags); // Send one byte
      Tx_ID++; // Switch to next message
      if (Tx_ID > 0x11) Tx_ID = 0x10; // Check overflow
      #endif
      #ifdef NODE2
      if (Rx_ID == 0x10) { // Check if this is our message
        PORTB = RxDx_Data[0]; // Display incoming data on PORTB
        RxDx_Data[0] = RxDx_Data[0] << 1; // Prepare data for sending back
        if (RxDx_Data[0] == 0) RxDx_Data[0] = 1; // Reinitialize if
        // maximum reached
      }
      Delay_ms(10); // Wait for a while
      CANSPIWrite(Tx_ID, &RxDx_Data, 1, Can_Send_Flags); // Send one byte
      // of data back
      }
      #endif
      #ifdef NODE3
      if (Rx_ID == 0x11) { // Check if this is our message
        PORTD = RxDx_Data[0]; // Display incoming data on PORTD
        RxDx_Data[0] = RxDx_Data[0] >> 1; // Prepare data for sending back
        if (RxDx_Data[0] == 0) RxDx_Data[0] = 128; // Reinitialize if
        // maximum reached
      }
      Delay_ms(10); // Wait for a while
      CANSPIWrite(Tx_ID, &RxDx_Data, 1, Can_Send_Flags); // Send one byte
      // of data back
      }
      #endif
    }
    else {
      // An error occurred, wait for a while
      #ifdef NODE1
      ErrorCount++; // Increment error indicator
      Delay_ms(10); // Wait for 100ms
      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, &RxDx_Data, 1, Can_Send_Flags);
        // Send new message
      }
      #endif
    }
  }
}
```

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.

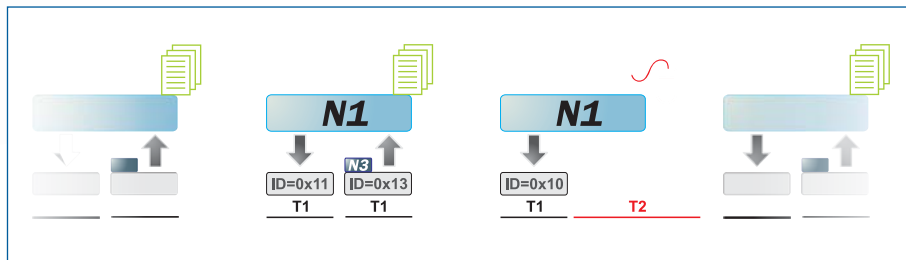


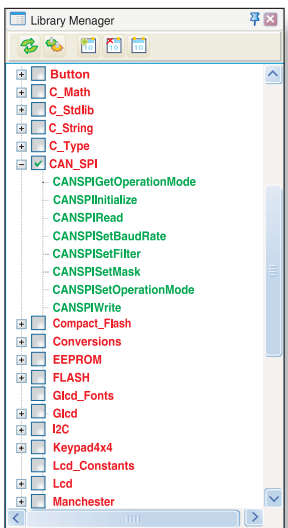
Figure 2. Exemple de communication.

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 PRO pour PIC® 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
 - CANISRead()* 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 PRO pour PIC utilisées dans le programme:
- Delay_us()
 - SPI1_init()
 - SPI1_read()



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

