MikroElektronika
DEVELOPMENT TOOLS | COMPILERS | BOOKS

OK. Brauchen Sie einen ... CAN-bus?

Drei *EasyPIC5_ Entwicklungssysteme* mit dem CAN-Bus über *CAN-SPI-Module* vernetzt

Manchmal ist es notwendig, dass man in einem System mehrere Mikrocontroller für unterschiedliche Aufgaben einsetzt und die Teile dennoch als ein Ganzes arbeiten müssen. In diesem Beitrag geht es darum zu zeigen, wie man drei Mikrocontroller an CAN anbindet und wie man Filter in CAN-Nodes benutzt, um Messages zu filtern.

Von Zoran Ristic
MikroElektronika Software-Entwicklung

Wenn mehrere Einheiten auf denselben Datenbus zugreifen, besteht immer die Notwendigkeit zu definieren, wie auf den Bus zugegriffen werden darf. Das CAN-Protokoll beschreibt präzise die Details der Verbindung mehrerer Einheiten zu einem Netzwerk und als solches wird es in der Industrie inzwischen breit eingesetzt. Das Protokoll definiert hauptsächlich die Priorität bei der Bus-Implementation und löst das Kollisions-Problem innerhalb der Hardware, falls mehrere Einheiten zur gleichen Zeit versuchen zu kommunizieren.

Hardware

In diesem Beispiel wird der CAN-Bus so konfiguriert, dass die erste Einheit Messages mit der ID 0x10 und 0x11 versendet, während die IDs der zweiten und dritten Einheit als 0x12 und 0x13 festgelegt werden. Die CAN-Nodes werden dann so konfiguriert, dass der zweite Node nur auf eingehende Messages mit dem Inhalt 0x10 antwortet, während die dritte Einheit dann auf den Wert 0x11 reagiert. Dazu passend reagiert der erste Knoten dann auf die beiden IDs 0x12 und 0x13 ID (Bild 2). Message-Filterung wird einfach implementiert, indem die Routine CANSPISetFilter aufgerufen wird, die dann alle nötigen Einstellungen der Mikrocontroller-Register und des CAN-SPI-Boards vornimmt.

Im Prinzip schreibt das CAN-Protokoll keinen Master zwingend vor. Doch um dieses Anwendungsbeispiel einfach und gut nachvollziehbar zu halten, wird hier nur der ersten Einheit als Master erlaubt, eine Kommunikation zu starten. Die beiden anderen Einheiten antworten lediglich auf individuellen Aufruf.

Software

Beim Senden einer Message lässt der Master-Node den aufgerufenen Nodes genug Zeit, um zu antworten. Bei einem Time-out (wenn diese Zeit verstrichen ist) meldet der Master einen Fehler und fährt mit der Kommunikation mit anderen Nodes fort (Bild 3). Falls ein peripherer CAN-Node zur selben Zeit wie ein anderer antwortet, entsteht eine „Kollision“ auf dem CAN-Bus. In diesem Fall verlangt die Device-Address-Priorität und der CAN-Bus, dass der Node mit der niedrigeren Priorität den Bus frei gibt, damit der Node mit der höheren Priorität seine Message sofort senden kann.

Wie schon erwähnt wird hier ein internes SPI-Modul des Mikrocontrollers dazu verwendet, Daten an den CAN-Bus zu übergeben. Einige der Vorteile der Verwendung des internen SPI-Moduls des Mikrocontrollers sind: Man kann beim Senden und Empfangen von Daten Interrupts generieren - das SPI-Module arbeitet dann unabhängig von der restlichen Peripherie. Außerdem ist es einfach zu konfigurieren. Die CAN-SPI-Library erlaubt die Festlegung des Arbeitsmodus von CAN-Bus und Node-Filtern, das Auslesen von Daten aus dem Buffer des CAN-SPI-Boards und vieles Anderes mehr.

In diesem Beispiel sind zusätzlich LEDs an die entsprechenden Pins des Mikrocontrollers geschaltet, mit denen sich

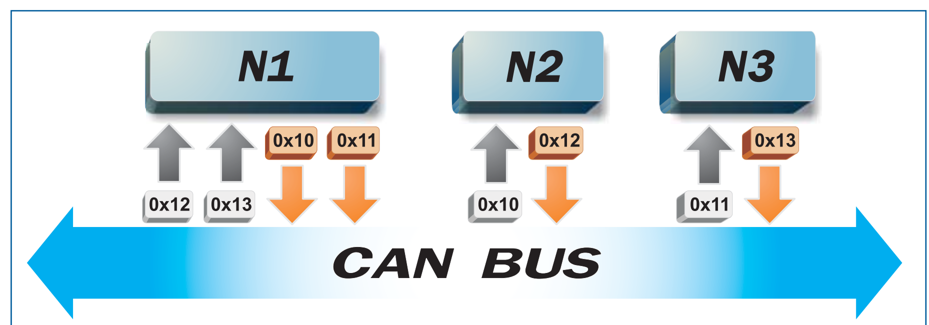
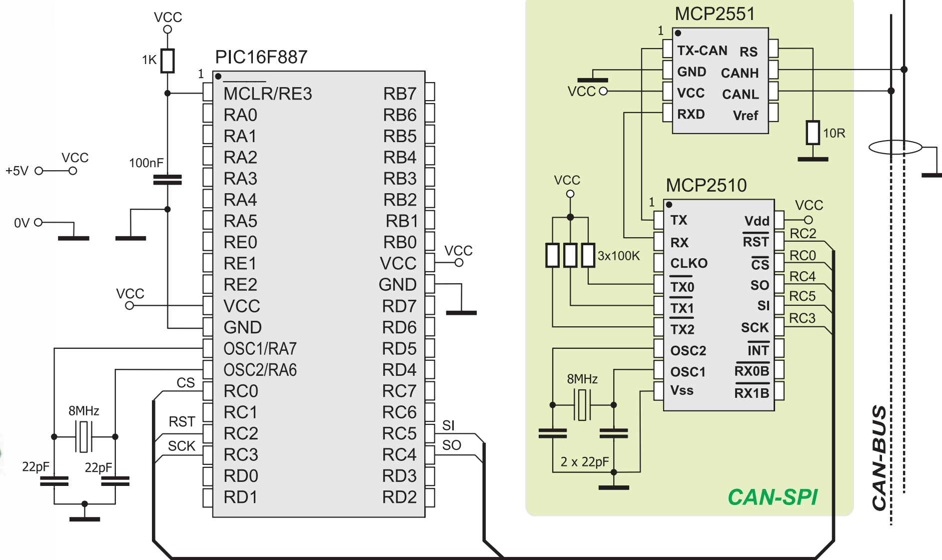


Bild 1. Message-Filterung.

Artikel von MikroElektronika www.mikroe.com
mikroC® und mikroC PRO® sind eingetragene Handelsmarken von MikroElektronika. Alle Rechte vorbehalten.



Schaltbild 1. Anschluss eines *CAN-SPI-Moduls* an einem PIC16F887.

die korrekte Funktion es Netzwerks überwachen lässt. Wenn Node 2 auf einen Aufruf von Node 1 antwortet, werden die LEDs an PORTB automatisch aktiviert. Wenn Node 3 auf den Aufruf antwortet, werden die LEDs an PORTD aktiviert. Der Source-Code für alle drei Nodes im Netzwerk wird hier vollständig wiedergegeben. Um für alle drei Nodes individuelle HEX-Files zu generieren, muss lediglich die Auskommentierung der DEFINE-Anweisungen im Beispiel-Header angepasst werden.

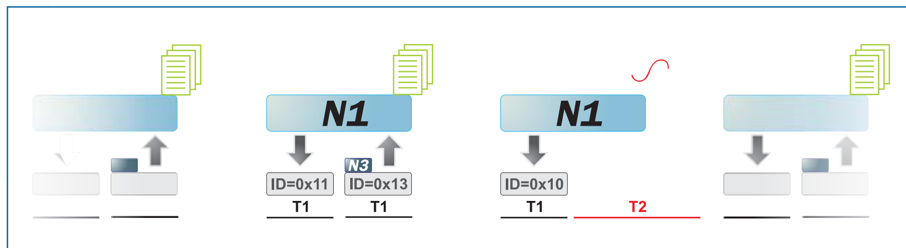
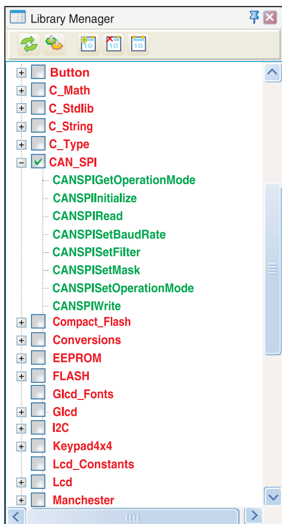


Bild 2. Beispiel-Kommunikation.

Zusammengefasst zeigt das Beispiel, wie man mehrere Mikrocontroller über den CAN-Bus koppelt. Auch die Frage des Umgangs mit Fehlern bezüglich des Kommunikations-Protokolls, falls ein angesprochener Node nicht oder nicht wie vorgesehen antwortet, wurde behandelt. Außerdem wurde gezeigt, wie Messages mit CAN-Filtern gefiltert werden können und wie Kommunikation auf dem Bus vonstatten gehen kann.

Library-Editor für ***mikroC PRO for PIC®*** mit anwendungsfertigen Libraries für: CAN_SPI, GLCD, Ethernet etc.



Im Programm verwendete Funktionen

CANSPIGetOperationMode()	Aktueller Arbeits-Modus
CANSPIInitialize()*	Initialisiere das CAN-SPI Module
CANIRead()*	Lese Message
CANSPISetBaudRate()	Setze CAN-SPI-Baudrate
CANSPISetFilter()*	Konfiguriere den Message Filter
CANSPISetMask()*	Erweiterte Filter Konfiguration
CANSPISet OperationMode()*	Aktueller Arbeits-Modus
CANSPIWrite()*	Schreibe Message

*** im Programm verwendete CANSPI-Library-Funktionen**

Andere im Programm verwendete Funktionen von *mikroC PRO for PIC*:

Delay_us()
SPI1_init()
SPI1_read()

GO TO Das Beispiel-Programm für PIC®-Mikrocontroller in den Sprachen C, BASIC und Pascal sowie die Software für dsPIC®, 8051- und AVR®-Mikrocontroller finden Sie auf unserer Webseite: www.mikroe.com/en/article/

Programm zur Demonstration der *CAN-Netzwerk-Funktion*.

```
#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 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;
// 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_XTD_FRAME &   //   with CANSPIWrite
                 _CANSPI_TX_NO_RTR_FRAME;

Can_Init_Flags = _CANSPI_CONFIG_SAMPLE_THRICE &  // Form value to
be used
                 _CANSPI_CONFIG_PHSEG2_PRG_ON &  //   with CANSPIInit
                 _CANSPI_CONFIG_XTD_MSG &
                 _CANSPI_CONFIG_DBL_BUFFER_ON &
                 _CANSPI_CONFIG_VALID_XTD_MSG;

SPI1_Init();                 // Initialize SPI module
CANSPIInitialize(1, 3, 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_XTD_MSG);
// Set all mask1 bits to ones
CANSPISetMask(_CANSPI_MASK_B2, -1, _CANSPI_CONFIG_XTD_MSG);
// Set all mask2 bits to ones
#ifdef NODE1
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
#else
CANSPISetFilter(_CANSPI_FILTER_B2_F2, 0x10, _CANSPI_CONFIG_XTD_MSG);
// Node2 and Node3 accept messages with ID 0x10
CANSPISetFilter(_CANSPI_FILTER_B1_F2, 0x11, _CANSPI_CONFIG_XTD_MSG);
// Node2 and Node3 accept messages with ID 0x11
#endif
CANSPISetOperationMode(_CANSPI_MODE_NORMAL,0xFF);  // Set NORMAL mode
RxTx_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, &RxTx_Data, 1, Can_Send_Flags);
// Node1 sends initial message
#endif
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
#ifdef NODE1
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
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 = RxTx_Data[0];     // Display incoming data on PORTB
RxTx_Data[0] = RxTx_Data[0] << 1; // Prepare data for sending back
if (RxTx_Data[0] == 0) RxTx_Data[0] = 1;   // Reinitialize if
                                           // maximum reached
Delay_ms(10);             // Wait for a while
CANSPIWrite(Tx_ID, &RxTx_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 = RxTx_Data[0];     // Display incoming data on PORTD
RxTx_Data[0] = RxTx_Data[0] >> 1;  // Prepare data for sending back
if (RxTx_Data[0] == 0) RxTx_Data[0] = 128; // Reinitialize if
                                           //maximum reached
Delay_ms(10);             // Wait for a while
CANSPIWrite(Tx_ID, &RxTx_Data, 1, Can_Send_Flags); // Send one byte
                                                   //of data back
         }
#endif
       }
else {                    // An error occured, 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, &RxTx_Data, 1, Can_Send_Flags);
// Send new message
         }
#endif
       }
    }
}
```

Microchip®, das Logo und seine Kombinationen sowie PIC® und andere sind eingetragene Handelsmarken der Microchip Corporation oder ihrer Töchter. Andere Begriffe oder Produktbezeichnungen können Handelsmarken anderer Firmen sein.