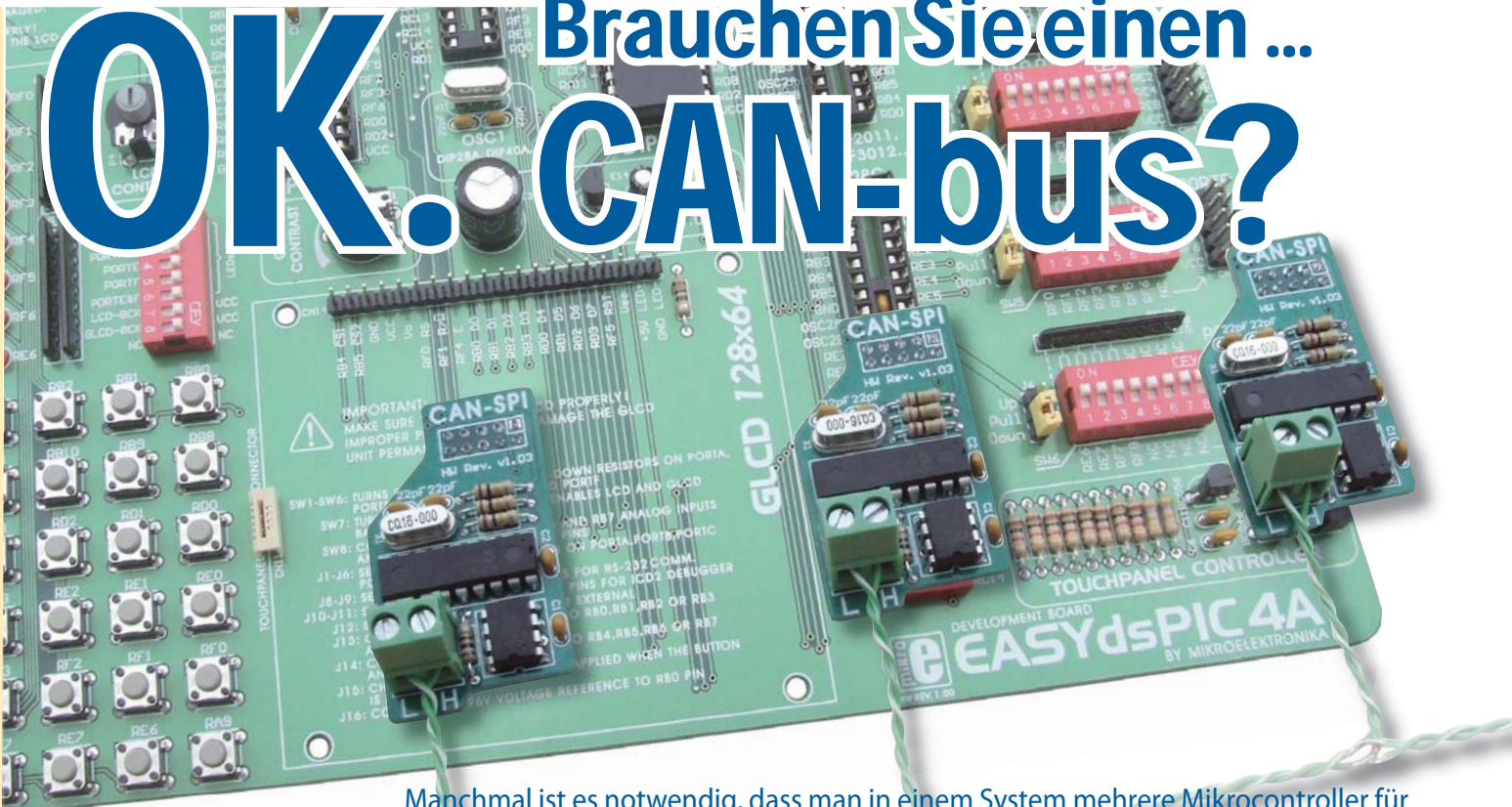


OK. CAN-bus?



EasysdsPIC4A Entwicklungssysteme und CAN-SPI-Module

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

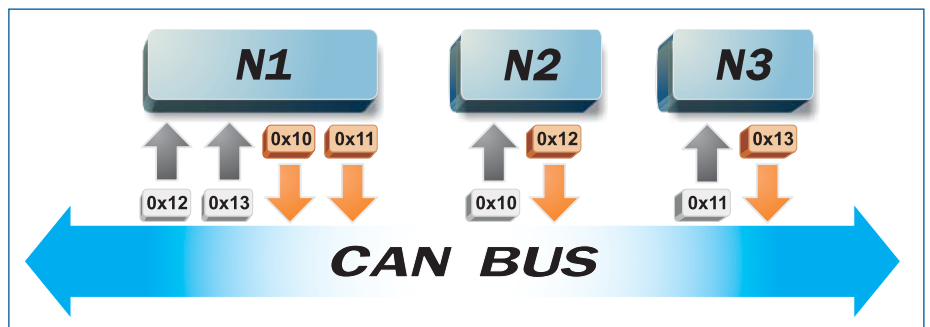
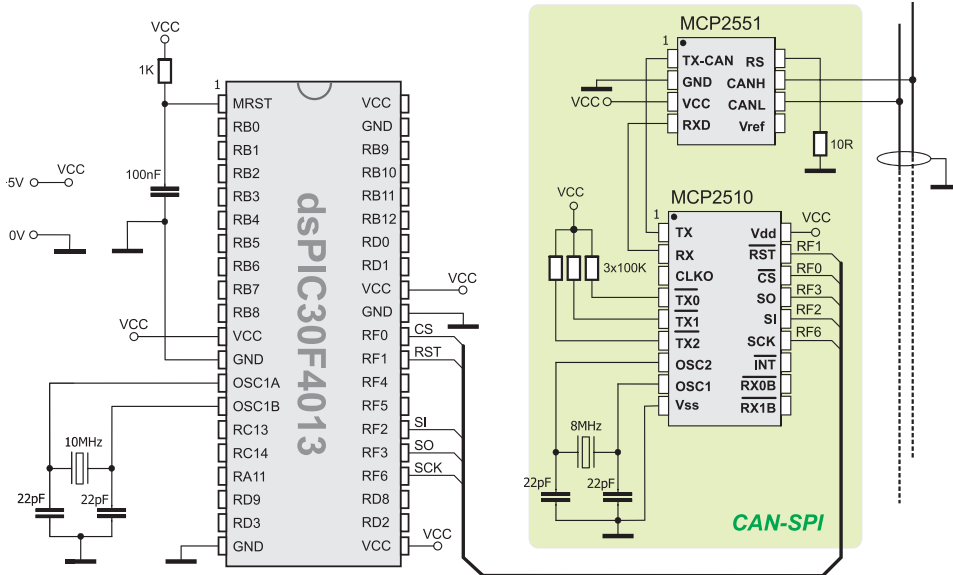


Bild 1. Message-Filterung.



Schaltbild 1. Anschluss eines CAN-SPI-Moduls an einem dsPIC30F4013.

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.

Programm zur Demonstration der CAN-Netzwerk-Funktion.

```

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();
    CANSPI1Init(1,3,3,1,Can_Init_Flags, &PORTF, 1, &PORTF,
    0);
    // Initialize external CANSPI module
    // initialize SPI module
    CANSPI1SetOperationMode(CANSPI_MODE_CONFIG, 0xFF);
    // set CONFIGURATION mode
    CANSPI1SetMask(CANSPI_MASK_B1, -1, CANSPI_CONFIG_XTD_
    MSG);
    // set all mask1 bits to ones
    CANSPI1SetMask(CANSPI_MASK_B2, -1, CANSPI_CONFIG_XTD_
    MSG);
    // set all mask2 bits to ones

    CANSPI1SetFilter(CANSPI_FILTER_B2_F4, 0x12, CANSPI_
    CONFIG_XTD_MSG);
    // Node1 accepts messages with ID 0x12
    CANSPI1SetFilter(CANSPI_FILTER_B1_F1, 0x13, CANSPI_
    CONFIG_XTD_MSG);
    // Node1 accepts messages with ID 0x13

    CANSPI1SetOperationMode(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

    CANSPI1Write(Tx_ID, &RxTx_Data, 1, Can_Send_Flags);
    // Node1 sends initial message

    while (1)
    // endless loop
    {
        Msg_Rcvd = CANSPI1Read(&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
            CANSPI1Write(Tx_ID, &RxTx_Data, 1, Can_Send_Flags
            );
            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
                CANSPI1Write(Tx_ID, &RxTx_Data, 1, Can_Send_
                Flags);
                // send new message
            }
        }
    }
}
    
```

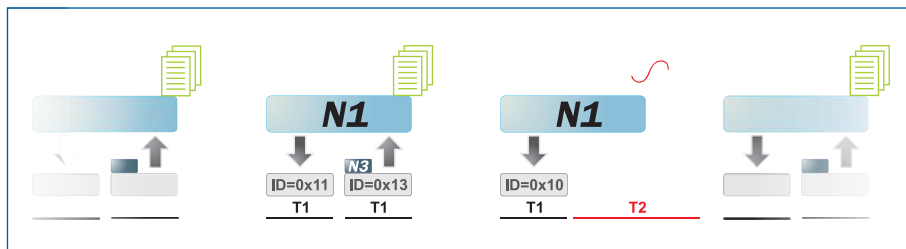


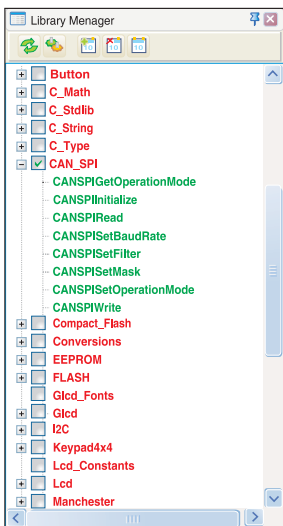
Bild 2. Beispiel-Kommunikation.

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 for dsPIC® 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 for dsPIC:
- Delay_us()
 - SPI1_init()
 - SPI1_read()



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

