

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



EasyAVR5A 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

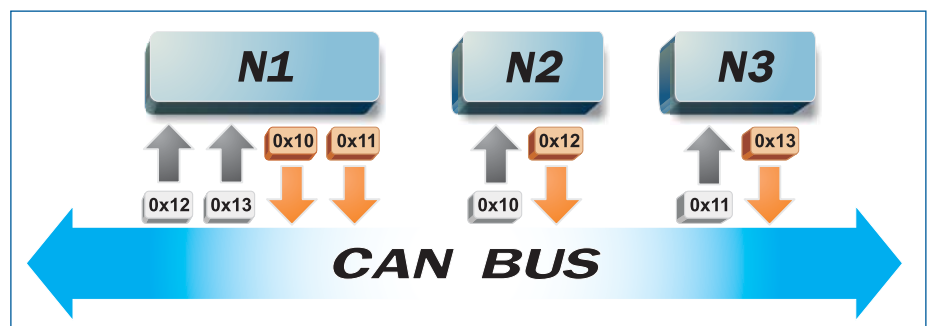
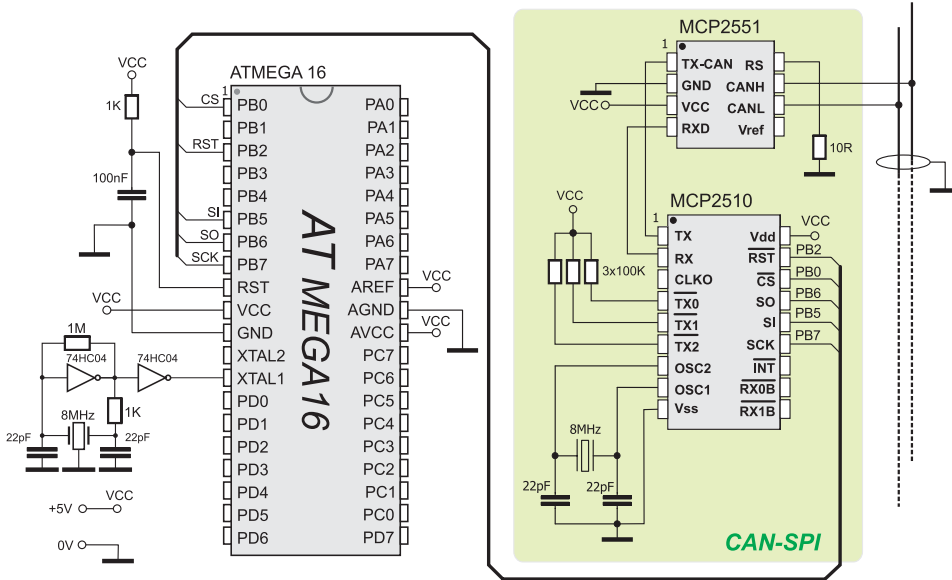


Bild 1. Message-Filterung.



Schaltbild 1. Anschluss eines CAN-SPI-Moduls an einem ATmega 16.

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 DEFIN-Anweisungen im Beispiel-Header angepasst werden.

Programm zur Demonstration der CAN-Netzwerk-Funktion.

```

program CanSPI
  ' Description: This program demonstrates how to make a CAN
  ' network using mikroElektronika
  ' CANSPI boards and mikroBasic compiler.
  ' Target device: ATMEGA 16
  ' Oscillator: 8MHz crystal

dim Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags as byte
' can flags
  Rx_Data_Len as byte
  ' received data length in bytes
  RxDx_Data as byte[8]
  ' can rx/tx data buffer
  Msg_Rcvd as byte
  ' reception flag
  Tx_ID, Rx_ID as longint
  ' can rx and tx ID
  ErrorCount as byte
' CANSPI module connections
dim CanSpi_CS as sbit at PORTB.B0
  ' Chip select (CS) pin for CANSPI board
  CanSpi_CS_Direction as sbit at DDRB.B0
  ' Direction register for CS pin
  CanSpi_Rst as sbit at PORTB.B2
  ' Reset pin for CANSPI board
  CanSpi_Rst_Direction as sbit at DDRB.B2
  ' Direction register for Reset pin
' End CANSPI module connections
main:
  ADCSRA.7 = 0
  ' Configure analog pins as digital I/O
  PORTB = 0
  DDRB = 255
  ' Initialize ports
  PORTD = 0
  DDRD = 255
  PORTC = 0
  DDRC = 255

  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 and
  ' form value to be used
  ' _CANSPI_TX_XTD_FRAME and
  ' with CANSPIwrite
  ' _CANSPI_TX_NO_RTR_FRAME

  Can_Init_Flags = _CANSPI_CONFIG_SAMPLE_THRICE and
  ' form value to be used
  ' _CANSPI_CONFIG_PHSB22_PRG_ON and
  ' with CANSPIInit
  ' _CANSPI_CONFIG_XTD_MSG and
  ' _CANSPI_CONFIG_DBL_BUFFER_ON and
  ' _CANSPI_CONFIG_VALID_XTD_MSG

  SPI1_Init()
  ' initialize SPI module
  CANSPIInitialize(1, 3, 3, 1, Can_Init_Flags)
  ' Initialize external CANSPI module
  CANSPISetOperationMode(_CANSPI_MODE_CONFIG, TRUE)
  ' 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
  RxDx_Data[0] = 0x40
  ' set initial data to be sent

  Tx_ID = 0x10
  ' set transmit ID for CAN message

  CANSPIWrite(Tx_ID, RxDx_Data, 1, Can_Send_Flags)
  ' Node1 sends initial message

  while (TRUE)
  ' endless loop
  Msg_Rcvd = CANSPIRead(Rx_ID, RxDx_Data, Rx_Data_Len,
  Can_Rcv_Flags) ' attempt receive message
  if (Msg_Rcvd) then
  ' if message is received then check id
  if Rx_ID = 0x12 then
  ' check ID
  PORTC = RxDx_Data[0]
  ' output data at PORTC
  else
  PORTD = RxDx_Data[0]
  ' output data at PORTD
  end if
  delay_ms(50)
  ' wait for a while between messages
  CANSPIWrite(Tx_ID, RxDx_Data, 1, Can_Send_Flags)
  ' send one byte of data
  inc(Tx_ID)
  ' switch to next message
  if Tx_ID > 0x11 then Tx_ID = 0x10 end if
  ' check overflow

  else
  ' an error occurred, wait for a while
  inc(ErrorCount)
  ' increment error indicator
  Delay_ms(10)
  ' wait for 10ms
  if (ErrorCount > 10) then
  ' timeout expired - process errors
  ErrorCount = 0
  ' reset error counter
  inc(Tx_ID)
  ' switch to another message
  if Tx_ID > 0x11 then Tx_ID = 0x10 end if
  ' check overflow
  CANSPIWrite(Tx_ID, RxDx_Data, 1, Can_Send_Flags)
  ' send new message
  end if

  end if
 wend
end.
  
```

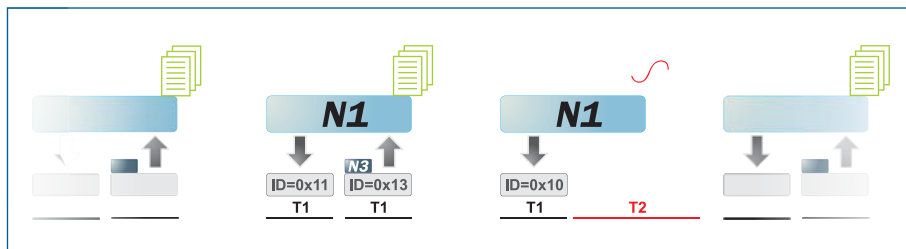
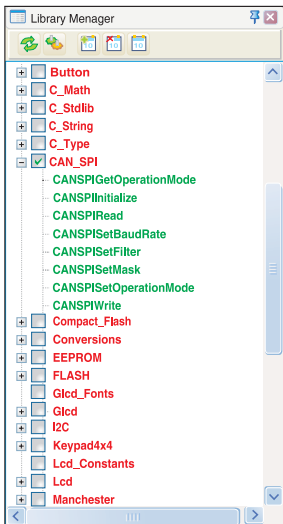


Bild 2. Beispiel-Kommunikation.

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 mikroBASIC PRO for AVR® mit anwendungsfertigen Libraries für: CAN\_SPI, GLCD, Ethernet etc.



Im Programm verwendete Funktionen

CANSPIGetOperationMode()	Aktueller Arbeits-Modus
CANSPIInitialize()*	Initialisiere das CAN-SPI Module
CANISRead()*	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 mikroBASIC PRO for AVR:  
 Delay\_us()  
 SPI1\_init()  
 SPI1\_read()

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

