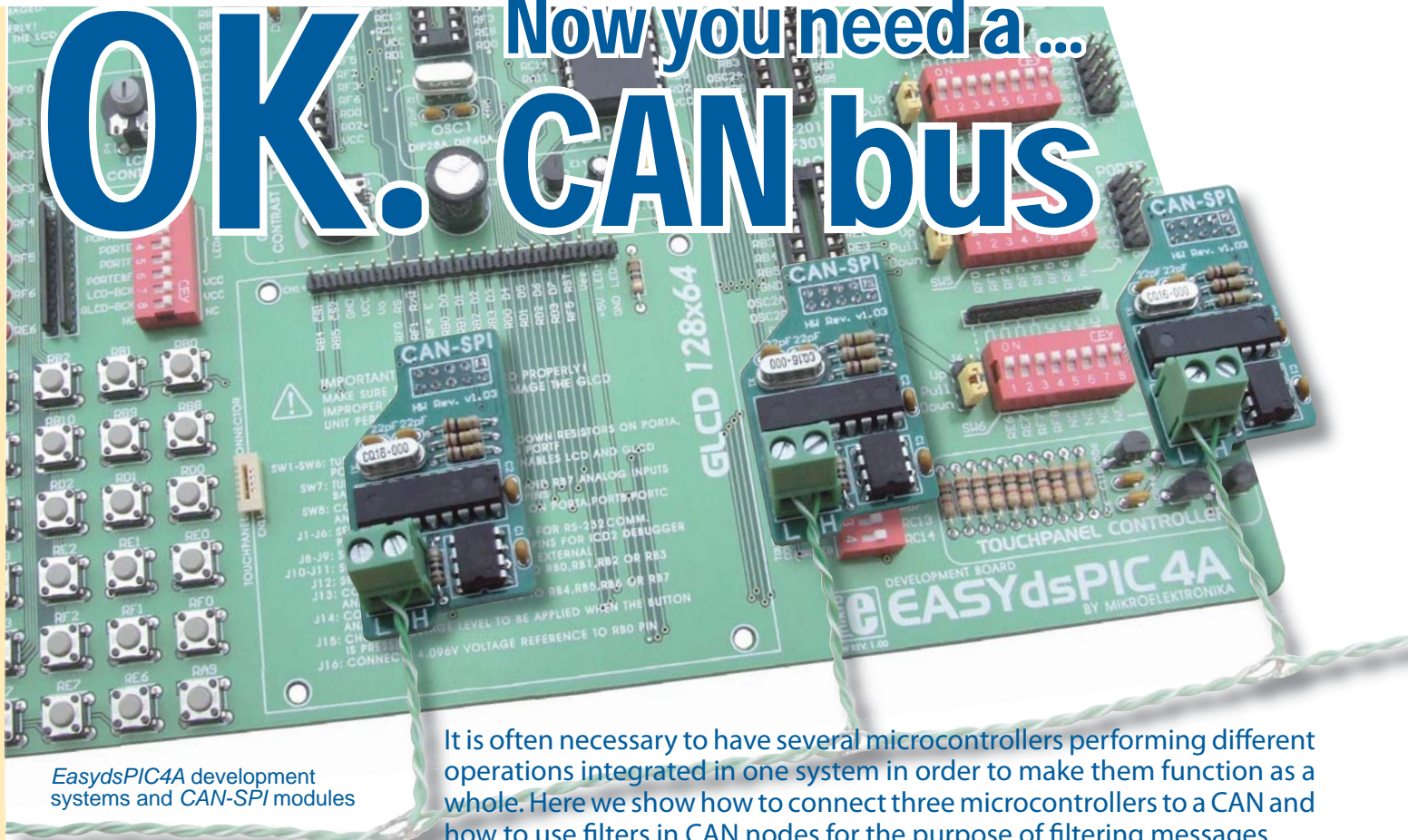


OK. Now you need a ... CAN bus



EasysPIC4A development systems and CAN-SPI modules

It is often necessary to have several microcontrollers performing different operations integrated in one system in order to make them function as a whole. Here we show how to connect three microcontrollers to a CAN and how to use filters in CAN nodes for the purpose of filtering messages.

By Zoran Ristic
MikroElektronika - Software Department

Whenever several peripheral units share the same data bus, it is necessary to define how the bus is accessed. The CAN protocol accurately describes all the details on connecting several devices to a network and as such it is widely used in the industry. The protocol primarily defines the precedence of bus implementation and solves the problem of 'collision' within the hardware in the event that several peripheral units start to communicate at the same time.

Hardware

In this example, a CAN bus will be configured so that the first device sends messages consisting of 0x10 and 0x11 as their ID, while the second and third device send messages consisting of IDs 0x12 and 0x13, respectively. We will also configure the CAN nodes so that the second node responds to incoming messages containing ID 0x10 only, while the third one responds only to those containing the 0x11 ID. Accordingly, the first device is configured to receive messages containing a 0x12 and 0x13 ID (Figure 2). Message filtering is easily implemented by calling the CANSPISetFilter routine which will also handle all the necessary settings of the microcontroller registers and CAN SPI board.

In general, the CAN protocol doesn't require a Master device to be present on the bus. However, to make this example easy to understand while still keeping it general-purpose, we will set the first device only, to initiate communication on the network and another two devices to respond to individual calls.

Software

When sending a message, the Master node leaves enough time for the called node to respond. In the event that a remote node doesn't respond within the time required, the Master reports an error in the current message and proceeds with calling other nodes (Figure 3). In the event that a peripheral CAN node responds at the same time as another node, a 'collision' will occur on the CAN bus. However, the device address prior-

ity and CAN alone prescribe that in this case the node transmitting the lower priority message withdraws from the bus, thus enabling the node transmitting the higher priority message to proceed with transmission immediately. As mentioned before, we will use an internal SPI module of the microcontroller to transfer data onto the CAN bus. Some of the advantages of using the microcontroller's internal SPI module are: the possibility of generating an interrupt when sending and receiving data; the SPI module operates independently of other peripherals and has a simple configuration. The CAN SPI library enables you to set the operating mode of the CAN and node filters, read data from the CAN SPI board buffer, etc.

This example also includes LEDs on the microcontroller pins indicating that the

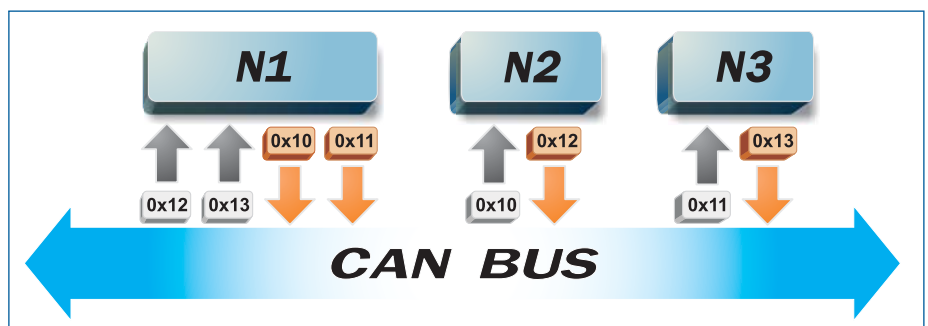
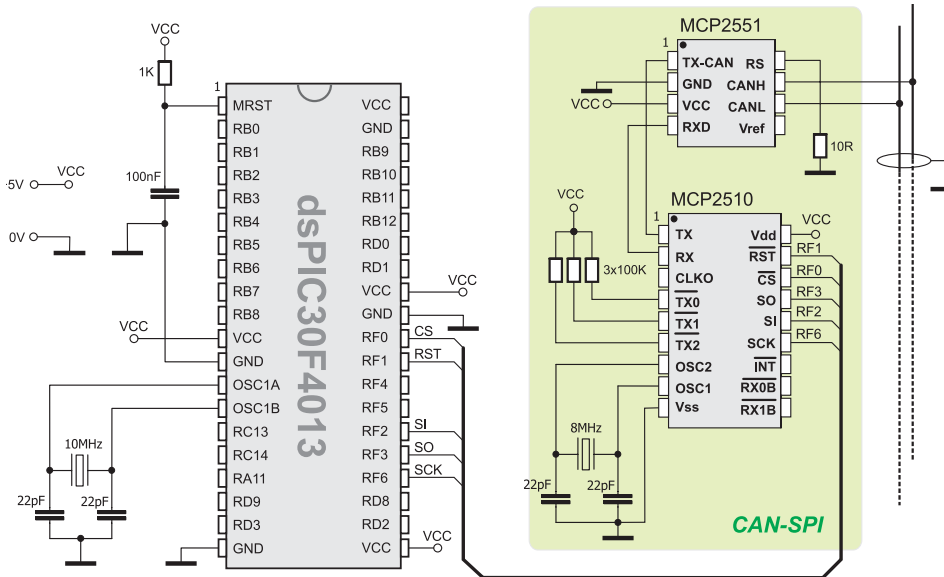


Figure 1. Message filtering



Schematic 1. Connecting the CAN-SPI module to a dsPIC30F4013

network operates properly. When node 2 responds to node 1's call, the PORTB LEDs will be automatically turned on. If node 3 responds to the call, the PORTD LEDs will be turned on. The source code for all three nodes in the network is provided with this example. In order to create a HEX file for each of these nodes individually, it is necessary to write only one DEFINE directive in the example header.

Program to demonstrate the operation of a CAN bus

```

program CanSPI
` Description: This program demonstrates how to make a
CAN network using mikroElektronika
`
` CANSPI boards and mikroBasic compiler for
dsPIC.
` Target device: Microchip dsPIC 30F4013
` Oscillator: 10MHz 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
RxTx_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

main:
ADPCFG = $FFFF
` Configure analog pins as digital I/O
PORTB = 0 TRISB = 0
` Initialize ports
PORTD = 0 TRISD = 0
TRISF = 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 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_PHSSEG2_PRG_ON and
` with CANSPIInit
CANSPI_CONFIG_XTD_MSG and
CANSPI_CONFIG_DBL_BUFFER_ON and
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, 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
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 (TRUE)
` endless loop
Msg_Rcvd = CANSPIRead(Rx_ID, RxTx_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
PORTB = RxTx_Data[0]
` output data at PORTB
else
PORTD = RxTx_Data[0]
` output data at PORTD
end if
delay_ms(50)
` wait for a while between messages
CANSPIWrite(Tx_ID, RxTx_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, RxTx_Data, 1, Can_Send_Flags)
` send new message
end if

end if
wend
end.
    
```

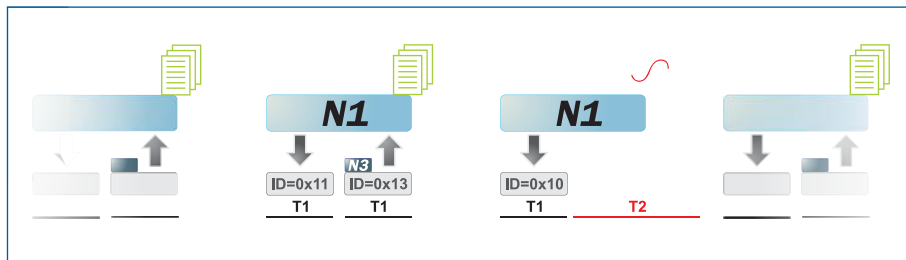


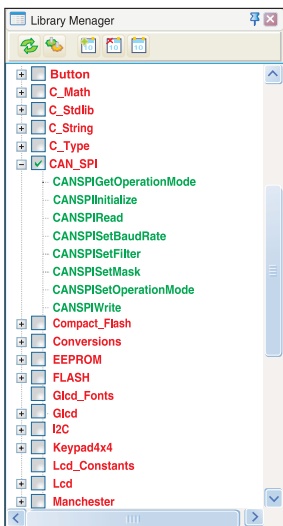
Figure 2. Communication example

In summary, here we have described one way of connecting microcontrollers to the CAN bus. We have also described how to detect errors by means of a communication protocol in the event that a remote node doesn't respond as expected, how to filter messages using CAN filters, as well as how to perform communication in general on the CAN bus.

mikroBASIC for dsPIC® library editor with ready to use libraries such as: CAN_SPI, GLCD, Ethernet etc.

Functions used in the program

CANSPIGetOperationMode()	Current operation mode
CANSPIInitialize()*	Initialize the CANSPI module
CANISRead()*	Read message
CANSPISetBaudRate()	Set the CANSPI baud rate
CANSPISetFilter()*	Configure message filter
CANSPISetMask()*	Advanced filtering configuration
CANSPISetOperationMode()*	Current operation mode
CANSPIWrite()*	Write message
* CANSPI library functions used in the program	
Other mikroBASIC for dsPIC functions used in the program:	
Delay_us()	
SPI1_init()	
SPI1_read()	



GO TO The program for this example written for dsPIC® microcontrollers in C, Basic and Pascal as well as the programs written for PIC® and AVR® microcontrollers may be found on our web site: www.mikroe.com/en/article/

