OK. Now you need a... CAN bus

Easy8051B 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 priority 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

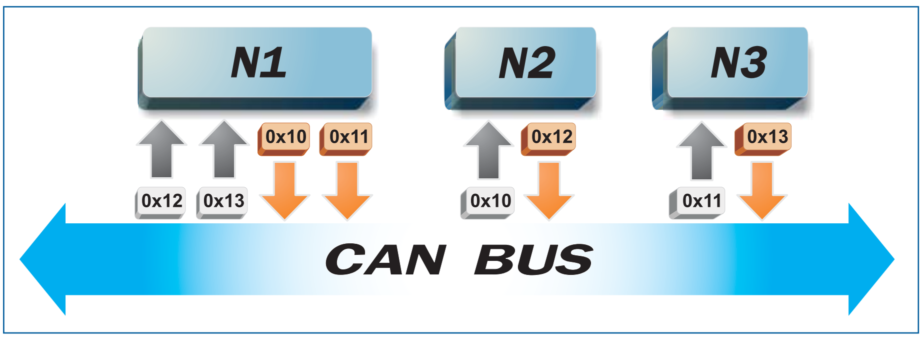
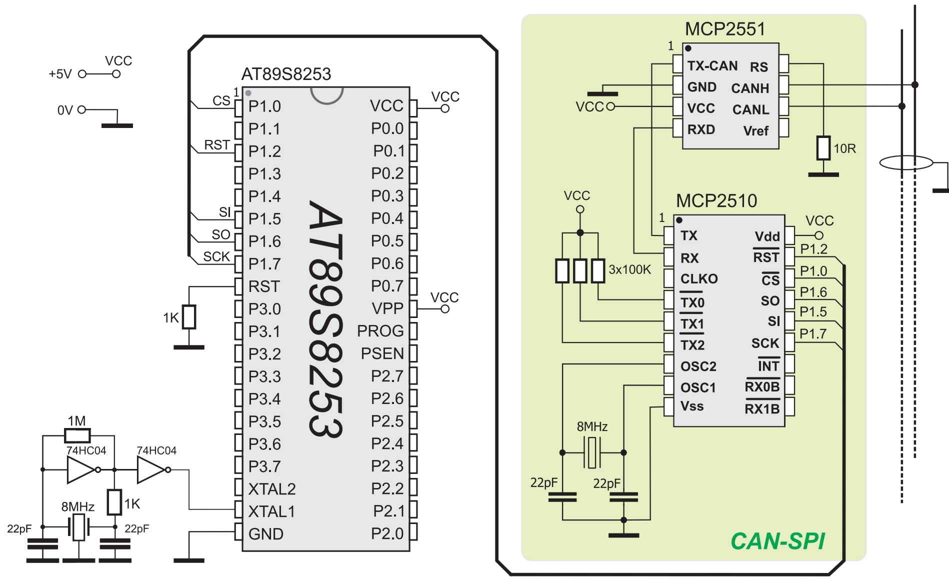


Figure 1. Message filtering



Schematic 1. Connecting the *CAN-SPI* module to a AT89S8253

the microcontroller pins indicating that the 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.

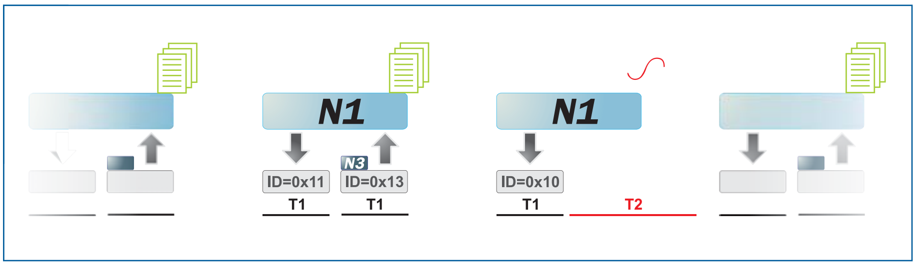
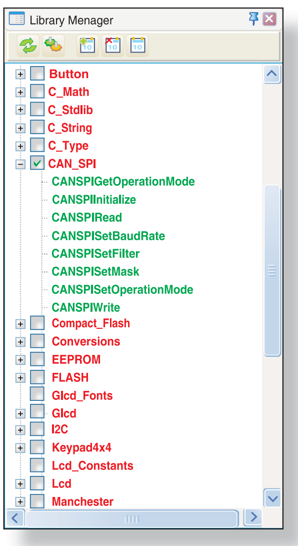


Figure 2. Communication example

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.

mikroPASCAL for 8051® 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
CANIRead()*	Read message
CANSPISetBaudRate()	Set the CANSPI baud rate
CANSPISetFilter()*	Configure message filter
CANSPISetMask()*	Advanced filtering configuration
CANSPISet OperationMode()*	Current operation mode
CANSPIWrite()*	Write message

*** CANSPI library functions used in the program**

Other *mikroPASCAL for 8051* functions used in the program:
Delay_us()
SPI1_init()
SPI1_read()

GO TO The program for this example written for 8051® microcontrollers in C, Basic and Pascal as well as the programs written for PIC®, dsPIC® and AVR® microcontrollers may be found on our web site: **www.mikroe.com/en/article/**

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 mikroPascal
compiler.
  Target device:  AT89S8253
  Oscillator:     8MHz crystal }

var Can_Init_Flags, Can_Send_Flags, Can_Rcv_Flags :
byte;                       // can flags
    Rx_Data_Len : byte;
                            // received data length in bytes
    RxTx_Data   : array[8] of byte;
                            // can rx/tx data buffer
    Msg_Rcvd : byte;
                            // reception flag
    Tx_ID, Rx_ID : longint;
                            // can rx and tx ID
    ErrorCount: byte;
// CANSPI module connections
var CanSpi_CS  : sbit at  P1.B0;
                            // Chip select (CS) pin for
CANSPI board
    CanSpi_Rst : sbit at  P1.B2;
// End CANSPI module connections
begin
  ErrorCount     := 0;
                           // Error flag
  Can_Init_Flags := 0; Can_Send_Flags := 0; Can_Rcv_
Flags := 0;                // clear flags

  Can_Send_Flags := CAN_TX_PRIORITY_0 and
                           // form value to be used
                    CAN_TX_XTD_FRAME and
                           // with CANSPIWrite
                    CAN_TX_NO_RTR_FRAME;

  Can_Init_Flags := CAN_CONFIG_SAMPLE_THRICE and
                           // form value to be used
                    CAN_CONFIG_PHSEG2_PRG_ON and
                           // with CANSPIInit
                    CAN_CONFIG_XTD_MSG and
                    CAN_CONFIG_DBL_BUFFER_ON and
                    CAN_CONFIG_VALID_XTD_MSG;

  Spi_Init();
                           // initialize SPI module
  CANSPIInitialize(1, 3, 3, 3, 1, Can_Init_Flags);
                           // Initialize external CANSPI module
  CANSPISetOperationMode(CAN_MODE_CONFIG, TRUE);
                           // set CONFIGURATION mode
  CANSPISetMask(CAN_MASK_B1, -1, CAN_CONFIG_XTD_MSG);
                           // set all mask1 bits to ones
  CANSPISetMask(CAN_MASK_B2, -1, CAN_CONFIG_XTD_MSG);
                           // set all mask2 bits to ones

  CANSPISetFilter(CAN_FILTER_B2_F4, 0x12, CAN_CONFIG_
XTD_MSG);                  // Node1 accepts messages with
ID 0x12
  CANSPISetFilter(CAN_FILTER_B1_F1, 0x13, CAN_CONFIG_
XTD_MSG);                  // Node1 accepts messages with
ID 0x13

  CANSPISetOperationMode(CAN_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) do
                           // endless loop
    begin
      Msg_Rcvd := CANSPIRead(Rx_ID, RxTx_Data, Rx_
Data_Len, Can_Rcv_Flags); // attempt receive message
      if (Msg_Rcvd) then begin
                              // if message is received
then check id

          if Rx_ID = 0x12 then
                           // check ID
              P0 := RxTx_Data[0]
                           // output data at PORT0
          else
              P2 := RxTx_Data[0];
                           // output data at PORT2
          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;
                           // check overflow

        end
      else begin
                           // an error occured, wait for a while

          inc(ErrorCount);
                           // increment error indicator
          Delay_ms(10);
                           // wait for 10ms
          if (ErrorCount > 10) then begin
                           // timeout expired - process errors
            ErrorCount := 0;
                           // reset error counter
            inc(Tx_ID);
                           // switch to another message
            if Tx_ID > 0x11 then Tx_ID := 0x10;
                           // check overflow
            CANSPIWrite(Tx_ID, RxTx_Data, 1, Can_
Send_Flags);               // send new message
          end;

        end;
    end;
end.
```

Written in compiler mikroPASCAL for 8051